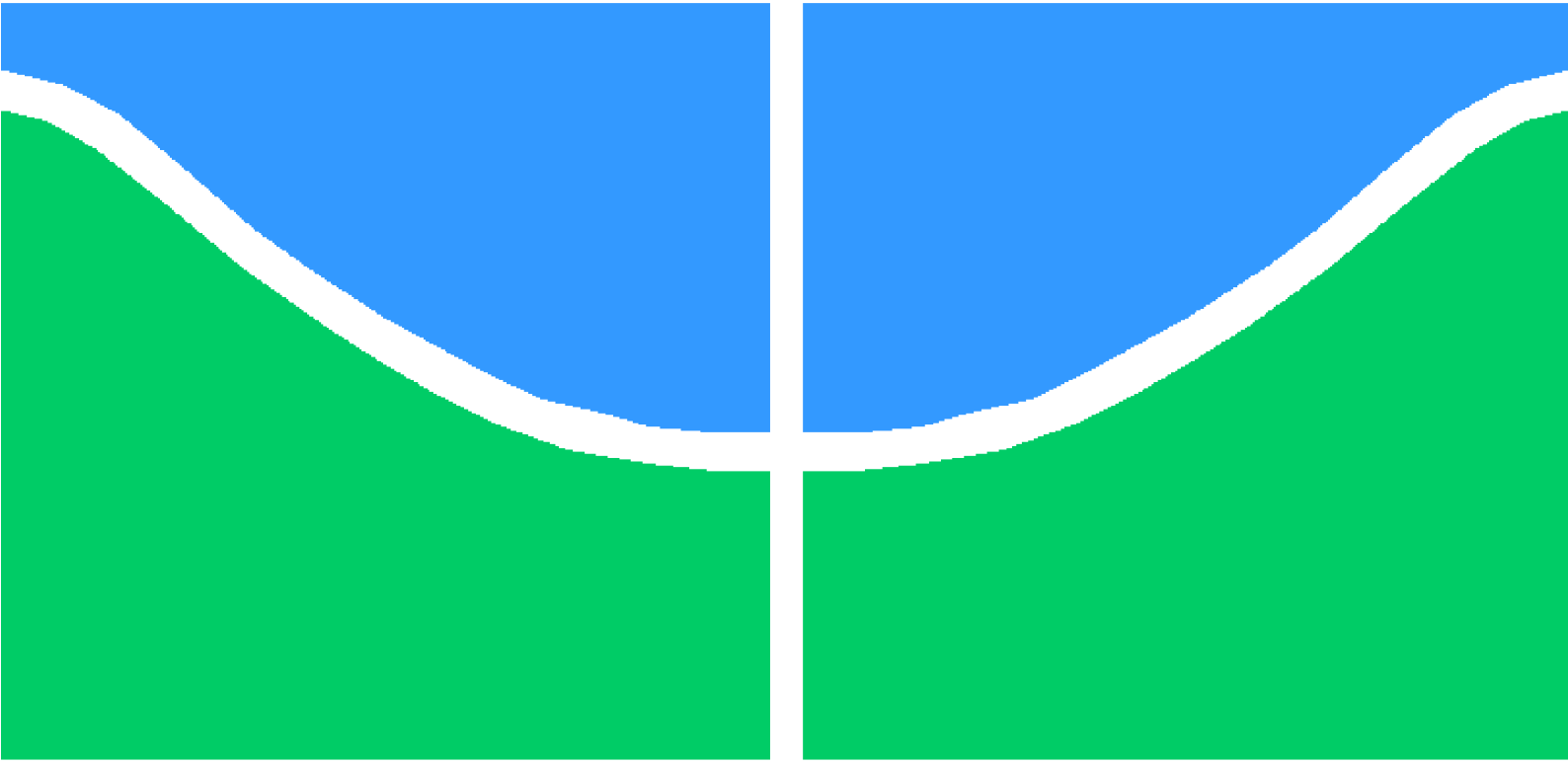


Universidade de Brasília - UnB
Departamento de Engenharia Elétrica
Engenharia Elétrica

Caixa Preta para Veículos Automotivos

Autor: Daniel Serra e João Victor Romualdo
Orientador: Prof. Ricardo Zelenovsky, Doutor

Brasília, DF
2017

Daniel Serra e João Victor Romualdo

Caixa Preta para Veículos Automotivos

Monografia submetida ao curso de graduação em Engenharia Elétricada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Elétrica.

Universidade de Brasília - UnB

Departamento de Engenharia Elétrica

Orientador: Prof. Ricardo Zelenovsky, Doutor

Brasília, DF

2017

Daniel Serra e João Victor Romualdo
Caixa Preta para Veículos Automotivos/ Daniel Serra e João Victor Romualdo. – Brasília, DF, 2017-
111 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Ricardo Zelenovsky, Doutor

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Departamento de Engenharia Elétrica , 2017.

1. IMU. 2. GPS. I. Prof. Ricardo Zelenovsky, Doutor. II. Universidade de Brasília. III. ENE/FT/UnB IV. Caixa Preta para Veículos Automotivos

CDU 02:141:005.6

Daniel Serra e João Victor Romualdo

Caixa Preta para Veículos Automotivos

Monografia submetida ao curso de graduação em Engenharia Elétricada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Elétrica.

Trabalho aprovado. Brasília, DF, 05 de dezembro de 2017:

Prof. Ricardo Zelenovsky, Doutor
Orientador

Prof. Alexandre Romariz, Doutor
Convidado 1

Vinícius de Oliveira Lima, Mestre
Convidado 2

Hudson Pereira Ramos, Bacharel
Convidado 3

Brasília, DF
2017

"Em terra de Saci,
qualquer chute é voadora...
(Desconhecido)

Resumo

O objetivo desse trabalho é o desenvolvimento de um equipamento de baixo custo que fornece todas as informações necessárias para a reconstrução de um acidente automotivo, para que, desta forma, exista um maior grau de confiabilidade e se possa gastar menos tempo e recursos com perícias. Esse sistema é a combinação de um hardware para a captação dos dados do acidente, que será composto de um módulo de IMU de 9 graus de liberdade e um módulo de GPS u-blox e de um software, dedicado ao processamento dos dados obtidos e a reconstrução do evento.

Palavras-chaves: IMU. Acidentes automotivos. Perícia técnica.

Abstract

Our goal with this project is to develop a low-cost equipment which records all the information needed to rebuild an accident scenario, increasing the reliability and lowering the operational costs of foresinc analysis. This system is a combination of a data-recording hardware, which is composed of a IMU module with 9 degrees of freedom and a GPS u-blox module. The simulation software is dedicated to the processing of the acquired data and the event reconstruction.

Key-words: IMU. Automotive accidents. Foresincs.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

DPVAT	Danos Pessoais Causados por Veículos Automotores de Via Terrestre
SUS	Sistema Único de Saúde
IPEA	Instituto de Pesquisa Econômica Aplicada
WHO	Organização Mundial da Saúde
EDR	Gravador de Dados de Evento
NHTSA	Administração Nacional de Segurança em Estradas e Trafego
NASA	Administração Nacional de Aeronáutica e Espaço
AEB	Freio Automático de Emergência
CONTRAN	Conselho Nacional de Trânsito
IDE	Ambiente de Desenvolvimento Integrado
ADC	Conversos Analógico-Digital
IMU	Unidade de Mensuração Inercial
MARG	Magnético, Taxa Angular, Gravidade
MEMS	Sensor Micro-Eletrônico-Mecânico
GPS	Sistema Global de Posicionamento
DOP	Diluição da Precisão
DGPS	GPS diferencial
TOW	Tempo da Semana
ECEF	Centrado na Terra, Fixo na Terra
UART	Transmissor-Receptor Universal Assíncrono
I2C	Circuito Inter-Integrado
SPI	Interface Serial Periférica
RMS	Raiz Média Quadrada

Sumário

Introdução

0.1 Contexto

O Brasil é um dos países do mundo em que mais ocorrem acidentes de trânsito. Segundo dados da Organização Mundial da Saúde (WHO, 2017), o Brasil teve 46.935 mortes em estradas apenas em 2013, e esses números vêm crescendo, como podemos ver no gráfico a seguir, que mostra o comparativo de 2003 até 2015 do número de indenizados por invalidez do DPVAT e o número de internações no SUS por acidentes de trânsito.

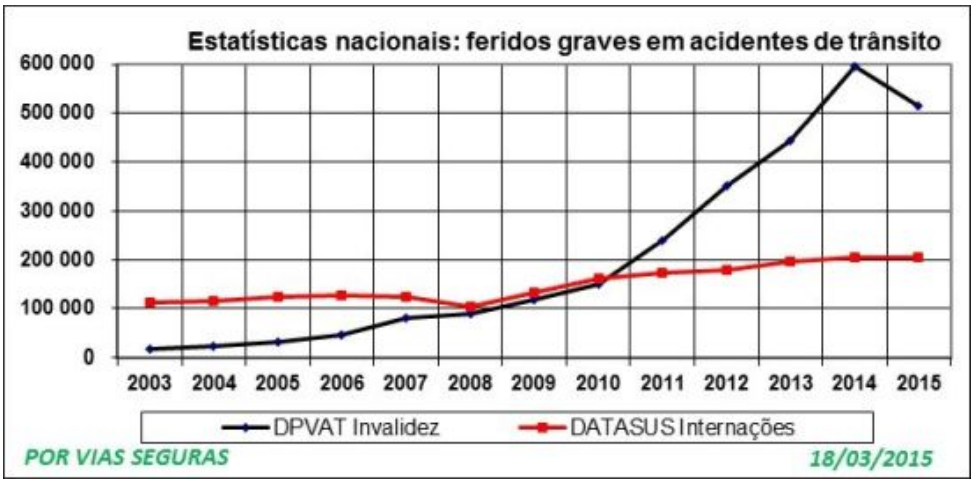


Figura 1 – Indenizados no DPVAT por invalidez e números de internações no SUS por acidentes de trânsito de 2003 até 2015

[FONTE: Vias Seguras]

Os números são ainda mais assustadores quando falamos de acidentes com vítimas fatais. De 2007 até 2010 os acidentes com morte aumentaram 22,23% de acordo com dados do Anuários Estatísticos das Rodovias Federais. Analisando acidentes rodoviários em geral, o número de mortes passa de 37.407 em 2007 para 40.610 em 2010, um salto de 8,56% "(Tabela 0.1)".

Na "Tabela (0.1)" temos os dados da quantidade de mortes decorrentes de acidentes rodoviários do Brasil, de alguns países desenvolvidos (Alemanha e Itália, por exemplo), do Chile e da Argentina. Dentre os citados na tabela, todos, com exceção do Brasil, tiveram uma redução no número de mortes, com méritos ao Reino Unido que reduziu de 3.059 (2007) para 1.850 (2010), uma queda de 39,52% .

Quando comparamos as frotas dos países citados fica mais claro ainda que o trânsito no Brasil é sem dúvidas um dos mais violentos do mundo. Nos Estados Unidos, por exemplo, havia mais de 274 milhões de veículos registrados no ano de 2014, contra pouco

Número de Mortes Decorrentes de Acidentes Rodoviários								
Ano	**Brasil**		**Estados Unidos**		**França**		**Chile**	
	Mortes	Variação	Mortes	Variação	Mortes	Variação	Mortes	Variação
2007	37.407	-	41.259	-	4.620	-	1.645	-
2008	38.273	2,32%	37.423	-9,30%	4.275	-7,47%	1.782	8,33%
2009	37.594	-1,77%	33.808	-9,66%	4.273	-0,05%	1.508	-15,38%
2010	40.610	8,02%	32.788	-3,02%	3.992	-6,58%	1.595	5,77%
Total	**153.884**	**8,56%**	**145.278**	**-20,53%**	**17.160**	**-13,59%**	**6.530**	**-3,04%**
Ano	**Reino Unido**		**Alemanha**		**Itália**		**Argentina**	
	Mortes	Variação	Mortes	Variação	Mortes	Variação	Mortes	Variação
2007	3.059	-	4.949	-	5.131	-	8.104	-
2008	2.645	-13,53%	4.477	-9,54%	4.725	-7,91%	8.205	1,25%
2009	2.337	-11,64%	4.152	-7,26%	4.237	-10,33%	7.885	-3,90%
2010	1.850	-20,84%	3.648	-12,14%	4.090	-3,47%	7.659	-2,87%
Total	**9.891**	**-39,52%**	**17.226**	**-26,29%**	**18.183**	**-20,29%**	**31.853**	**-5,49%**

Tabela 1 – Mortes decorrentes de acidentes rodoviários em oito países, de 2007 a 2010

mais de 86 milhões que rodam em estradas muitas vezes não pavimentadas. Ou seja, com uma frota 3 vezes maior, nos E.U.A. ocorreram 29.989 mortes no trânsito contra 43.780 no Brasil no ano de 2014.

Toda essa insegurança nas estradas gera um custo elevado ao país. O IPEA estima que só em 2014 os gastos com acidentes em rodovias federais, estaduais e municipais podem superar os R$40 bilhões. Isso sem contar com as perdas irreparáveis das famílias dos acidentados e a perda de produtividade da força de trabalho como consequência das mortes e invalidez.

0.2 Motivação

A ideia de se fazer um dispositivo análogo ao já presente em aviões, a caixa-preta, é para auxiliar os profissionais de segurança pública em perícias de acidentes. O conhecimento das causas e das consequências de acidentes automotivos fornece pistas para medidas que possam vir a melhorar o atual cenário brasileiro de violência nas estradas.

Hoje em dia as perícias são baseadas em vestígios encontradas no local do acidente. Seja a deformação estrutural do veículo, marcas na pista (pneu, metal, fluidos, etc.), danos na infraestrutura da estrada ou testemunhas, todas essas bases de informação do perito podem ser deterioradas se o local não for preservado ou o tempo de chegada da polícia científica for demasiadamente longo.

As caixas-pretas nos aviões são essenciais no entendimento de acidentes e também na prevenção dos mesmo, pois, uma vez conhecido um problema ou comportamento

problemático que gera um acidente, pode-se atuar na mitigação desses acontecimentos. Veículos modernos que já venham de fábrica com recursos mínimos de segurança, como por exemplo o *Airbag*, contém um dispositivo que controla o acionamento desse, e outros, equipamentos e que também armazena alguns dados que podem ser usados para entender momentos antes, durante e depois de acidentes, esse dispositivo é chamado de EDR.

Essa discussão da utilização de EDRs em veículos automotivos não é recente. Em 2008, foi proposto o projeto de lei 2868A (DCD, 2011), na Câmara dos Deputados, que tratava da obrigatoriedade de se instalar EDRs em todos os automóveis em território brasileiro. E esse assunto também não é restrito ao escopo brasileiro. Em 1997, a própria NASA recomendou ao NHTSA que estudassem a aplicabilidade desses dispositivos em carros (NHTSA, 2011).

Figura 2 – Perícia realizada em João Pessoa-PB para determinar velocidade de veículo envolvido em acidente (G1-PARAíBA, 2015)

Os automóveis novos já possuem EDRs incorporado ao seu sistema, desde 2014 com a decisão do CONTRAN. Em uma estimativa também da NHTSA, 96% dos veículos atuais saem com esses dispositivos já instalados de fábrica (RAFTER, 2014), nos EUA Esses módulos, todavia, são utilizados em sua maioria para tarefas mais simples, como a ativação de *Airbag* em batidas, sistemas AEB para evitar colisões, dentre outras funções. Um grande fator limitante da utilização desses módulos nativos é que não existe um padrão entre as fabricantes quanto a leitura das informações, dificultando muito uma engenharia reversa dos dados. Além disso, os kits para as obtenções desses dados são muito caros, estando na margem dos 2000 até os 10000 dólares americanos.

0.3 Objetivo

Com as informações acima em mente, temos por objetivo prototipar um dispositivo de baixo custo capaz de coletar e armazenar informações inerciais do veículo antes, durante e depois de uma colisão. Essas informações serão então processadas por um algoritmo e reproduzidas de forma a simular a dinâmica do acidente. Este projeto é uma continuação da tese de mestrado do Vinícius Lima e do trabalho de graduação de Hudson Pereira Ramos e Vanessa Oliveira Lucena

Esse dispositivo poderá ser usado em veículos oficiais de polícia para coletar dados de uma eventual colisão e assim esses dados serão usados na perícia do evento. Uma vez que o dispositivo tenha sido validado e sua efetividade no auxílio do trabalho de perícia seja comprovada, acreditamos que poderá ser proposto como política de segurança pública que veículos comercializados no Brasil venham com dispositivo com capacidades semelhantes às propostas nesse trabalho.

Como objetivo de longo prazo, esse trabalho propõe um método simples e barato de coletar dados inerciais que poderão ser usados para guiar políticas de segurança veicular desde a fabricação até a melhora na infraestrutura de estradas de rodagem no Brasil.

Parte I

Embasamento

1 Embasamento

Nesse capítulo descreveremos os principais elementos que compõe nossa plataforma inercial.

1.1 Sensores Inerciais e Rotacionais

Para esse projeto, utilizaremos o sensor inercial MARG MPU-9250, que é composto de um giroscópio, acelerômetro e magnetômetro, todos de 3 eixos, constituindo assim um sensor com nove graus de liberdade. Essa redundância será fundamental para corrigir erros causados pelo ruído presente em cada um desses elementos individuais.

Também será usado um sensor GPS u-blox NEO-6m. Esse receptor GPS tem um ótimo custo *versus* benefício e para nosso propósito é bastante adequado.

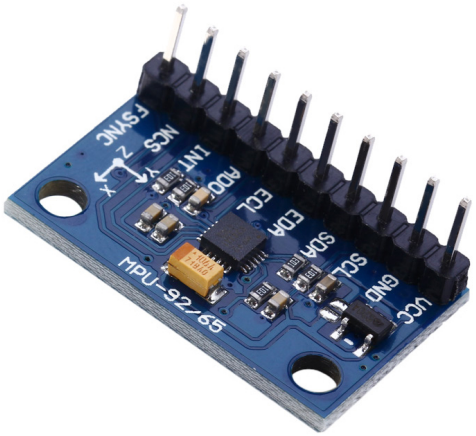


Figura 3 – Módulo MPU-9250

1.1.1 Acelerômetro

O acelerômetro é um dispositivo MEMS corda estendida com uma massa de prova em seu centro, posicionada de forma que essa massa possa sofrer a influência de qualquer aceleração imposta ao sistema em combinação com um transdutor para que se possua uma saída utilizável (SENSORWIKI).

Esse dispositivo mede a aceleração do aparato em relação à aceleração de queda livre (de módulo igual ao campo gravitacional, com sentido para baixo), ou seja, em suas medidas sempre terá um *offset* de módulo g (9.98 m/s, ao nível do mar) e sentido para cima, dessa forma, caso ele seja posto em queda livre, teremos que sua saída será nula.

Utilizaremos esse dispositivo para termos uma descrição bem precisa dos vetores de forças atuantes no carro no momento da batida e posteriormente.

1.1.2 Giroscópio

O giroscópio é um sensor que mede velocidade angular relativa a um eixo. Seu funcionamento é baseado no efeito da força de Coriolis. Essa força, por sua vez, é uma força inercial que atua em objetos que estão sendo vistos por um referencial inercial em rotação (ESCHOOLTODAY).

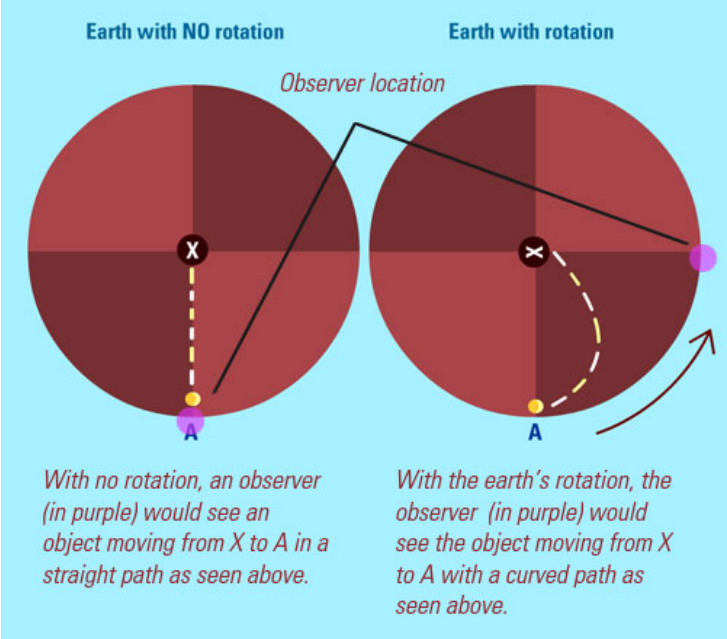


Figura 4 – Exemplo da Força de Coriolis (ESCHOOLTODAY)

A saída desse dispositivo é a medição dessa força aplicada a um elemento vibrante, de forma que ela é proporcional à velocidade de rotação do sistema como um todo (EPSON).

Esse sensor será utilizado para uma descrição detalhada de todas as rotações que ocorreram, desde as que foram causadas pela colisão em si, possíveis capotamentos, ou até derrapagens que ocorreram antes e que podem ter sido a causa do evento.

1.1.3 Magnetômetro

Um magnetômetro é um dispositivo capaz de medir alterações no campo magnético no local. Seu funcionamento é baseado na mudança do movimento dos elétrons e buracos em um semicondutor descrito por efeito Hall. Tendo essas correntes, é possível calcular a intensidade e a direção do campo magnético atuante (MABE, 2017).

Como uma bússola, esse dispositivo sofre com diversas interferências eletromagnéticas e sem uma boa calibração os dados desse sensor são inúteis. Uma subseção será dedicada a explicação da calibração do magnetômetro.

1.1.4 GPS

O Sistema Global de Posicionamento (do inglês, *Global Positioning System*) já está em funcionamento pleno desde 1993 (MAI, 2012), porém o programa teve início em 1973 como um projeto militar e o primeiro satélite foi lançado em órbita em 1978. Hoje a tecnologia de referenciamento já é acessível para qualquer pessoa que possua um *smartphone*.

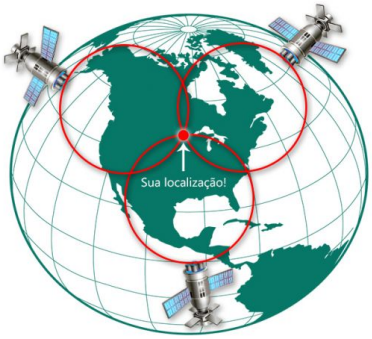


Figura 5 – Trilateração de sinais GPS

Seu funcionamento é baseado na trilateração (GRIFFIN, 2011) em três dimensões onde o receptor recebe, decodifica e determina a que distância se está de pelo menos quatro satélites. De forma bem simplificada, o receptor determina quanto tempo o sinal demorou para sair do satélite e ser recebido e multiplica esse tempo pela velocidade de propagação de ondas eletromagnéticas. Certamente, há diversas outras influências sobre como a posição é aferida de forma precisa, entretanto, tais detalhes fogem do escopo desse trabalho.

1.2 Microcontrolador

Nesse trabalho, um Arduino MEGA 2560 será usado para controlar todos os outros elementos e gerenciar o funcionamento da plataforma. O Arduino MEGA 2560 é baseado no microcontrolador AVR ATmega2560 da Atmel e sua programação pode ser feita pela IDE Arduino, o que permite desenvolvimento ágil e prático. Dentre as especificações do microcontrolador, destacamos essas (ATMEL):

- Velocidade de *clock* de 16MHz (4,5 5̃,5V);

- 256KB de memória de programa;
- 8KB de memória SRAM;
- 5KB de memória EEPROM;
- 54 pinos digitais;
- 16 pinos ADC;
- I2C nativo;
- SPI nativo;
- UART nativo.

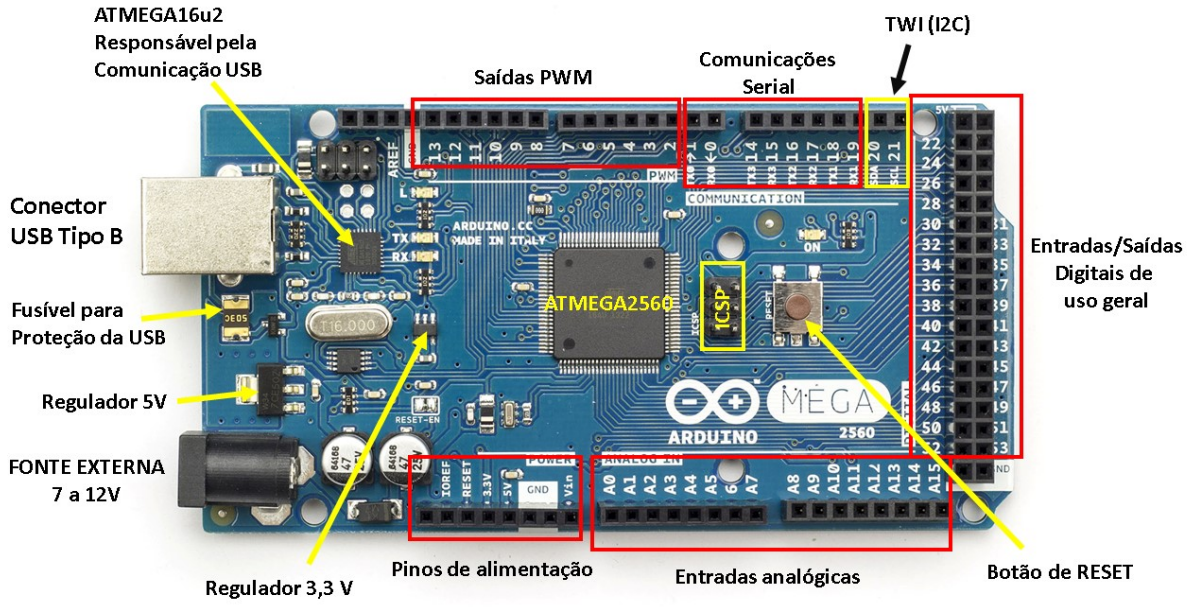


Figura 6 – Arduino Mega 2560 com indicativo de funcionalidades (SOUZA, 2014)

1.3 Protocolos de Comunicação

Para "conversar" com os diversos dispositivos presentes no protótipo, faz-se o uso de alguns padrões de comunicação bastante comuns.

1.3.1 UART

Universal asynchronous receiver-transmitter é uma forma de comunicação serial assíncrona muito comum em microcontroladores e normalmente *full-duplex*, onde é possível transmitir e receber simultaneamente. Nesse modo de comunicação, *bytes* são enviados

bit a *bit* de modo sequencial e contínuo (BASICS). Para funcionar em modo *full-duplex* é necessário três conexões: Terra, TX e RX, onde o TX de um dispositivo conecta-se ao RX do outro dispositivo e vice-versa e ambos são conectados ao mesmo Terra.

Para haver comunicação (um dispositivo entender o outro, e vice-versa) é preciso que ambos estejam em uma mesma configuração, ou seja, deve-se garantir que a *baudrate*, o número de *bit* de dados, a condição de parada (*stop bit*) e a paridade sejam iguais. Em nosso projeto, o GPS usa esse sistema de comunicação, configurado com 115.200 de *baudrate*, 8 *data bits*, sem paridade e 1 *stopbit*.

1.3.2 I^2C

Esse protocolo de comunicação desenvolvido pela Phillips chamado de *Inter-Integrated Circuit* é um padrão serial de comunicação desenvolvido para comunicar multi-dispositivos em um mesmo circuito, ou seja, próximos um do outro. O I2C usa um esquema mestre-escravo onde o mestre coordena toda a comunicação e é responsável por pedir dados aos escravos, assim como enviar dados para eles.

Para isso, usa se duas linhas: SDA (*Serial Data*) e SCL (*Serial Clock*) (REIS, 2014); diversos dispositivos podem ser ligados nesse barramento onde a limitação vem da possibilidade de endereços (cada escravo tem um endereço próprio) e da capacitância do barramento.

Em nosso projeto usa-se a comunicação I2C para configurar e coletar dados do IMU (MPU-9250) e também para escrever e ler na memória EEPROM.

1.3.3 SPI

O *Serial Peripheral Interface* é um protocolo serial síncrono desenvolvido pela Motorola nos anos 1980 muito utilizado em sistemas embarcados. É um tipo de comunicação *full-duplex* que também usa a arquitetura mestre-escravo com um único mestre (SACCO, 2014).

Esse sistema usa 4 linhas para comunicar dois dispositivos, onde 3 dessas linhas podem ser compartilhadas com outros escravos, são elas: *clock*, MISO e MOSI. A quarta linha é denominada SS. O *clock* é a linha de sincronismo, MOSI é a linha de dados no sentido mestre-escravo, MISO é a linha de dados escravo-mestre e o SS é por onde o mestre seleciona com qual escravo ele ira se comunicar. Nesse projeto a memória volátil SRAM usa esse esquema de troca de informações.

1.4 Memórias

O IMU fornece 9 dados de 16 bits, assim, cada leitura tem dezoito *bytes* de informação. A uma taxa de 100Hz, são 1800 *bytes* por segundo. Esse tanto de informação esgotaria a memoria RAM do Arduino MEGA em 4,44 segundos caso toda sua memória estivesse livre, o que na realidade não ocorre.

Para conseguirmos lidar com essa quantidade de dados é preciso usar uma memória externa que seja rápida e capaz de muitos ciclos de escrita. Por outro lado também precisamos de uma memória que possa reter esses dados quando em um evento de colisão onde a alimentação (energia) seja cortada. Para isso usamos dois tipos de memória que são descritas abaixo.

1.4.1 EEPROM

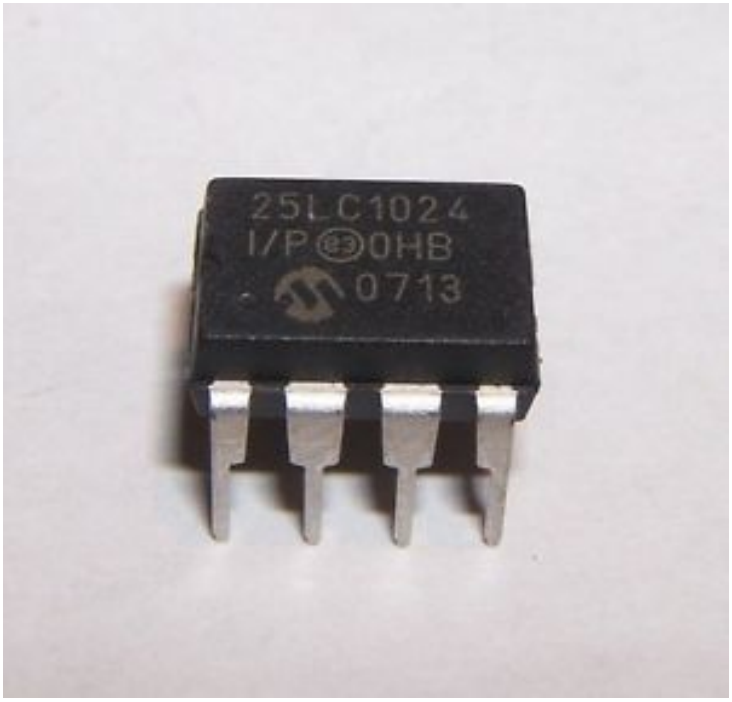


Figura 7 – Módulo de memória EEMPROM

A memória EEPROM (***e**lectrically **e**rasable **p**rogrammable **r**ead-**o**nly **m**emory*) é um tipo de memória não-volátil, ou seja, que não perde os dados quando é desligada da alimentação. Esse tipo de memória tem um ciclo limite de escritas/apagues que no nosso caso, 24FC1025 da Microchip, é de um milhão de ciclos.

Outras características dessa memória são: 128 kb de armazenamento divididos em dois blocos; retenção de dados por mais de 200 anos (MICROCHIP). Pode-se gravar *byte* a *byte* ou em páginas de 128 *bytes*; pode-se ler um único *byte* ou sequencialmente até o limite do bloco.

1.4.2 SRAM

A memória SRAM (*Static random-access memory)* é uma memória volátil que usa *flip-flops D* para armazenar cada bit. A latência de escrita dessa memória é muito baixa, tornando-a uma excelente memória para nossa aplicação. Outro aspecto positivo é ilimitação de ciclos de leitura e escrita.

Nesse projeto usamos a memória 23FC1024, que armazena pouco mais que 1 milhão de bits organizados em 4096 páginas de 32 *bytes*(MICROCHIP). É possível escrever e ler desde um *byte* de cada vez até a todo o conteúdo de uma só vez. Em nossa plataforma, essa memória será usada de forma cíclica, onde todo seu conteúdo será reescrito a cada aproximadamente 70 segundos.

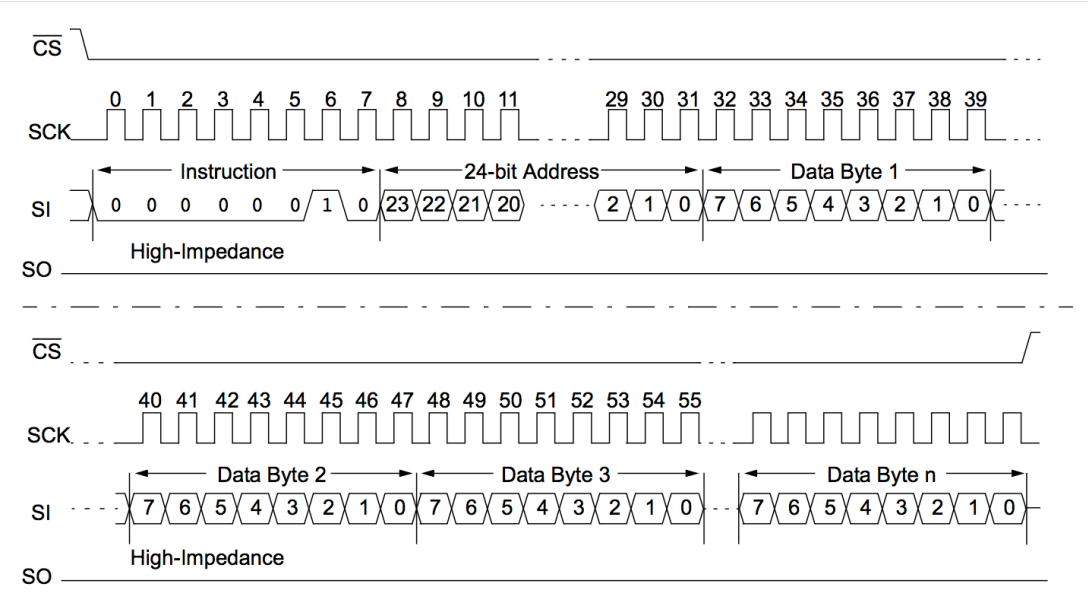


Figura 8 – Sequência de instruções para escrita sequencial de dados na SRAM (MICROCHIP)

1.5 Calibração do Magnetômetro

A calibração do magnetômetro é um processo complexo e muito passível de erros, já que o campo magnético da Terra é medido em valores muito pequenos. Para que este procedimento seja adequadamente realizado, deve-se entender a natureza dos erros a serem eliminados.

A figura a seguir mostra as medições dos três eixos do magnetômetro(Mx, My e Mz) para o caso de um magnetômetro não calibrado. O semi-eixo de cada elipse simboliza a sensibilidade da leitura do magnetômetro para cada eixo, e o centro do círculo indica um *offset* da leitura. Idealmente, os três círculos deveriam ser concêntricos e com o mesmo raio (WINER, 2017).

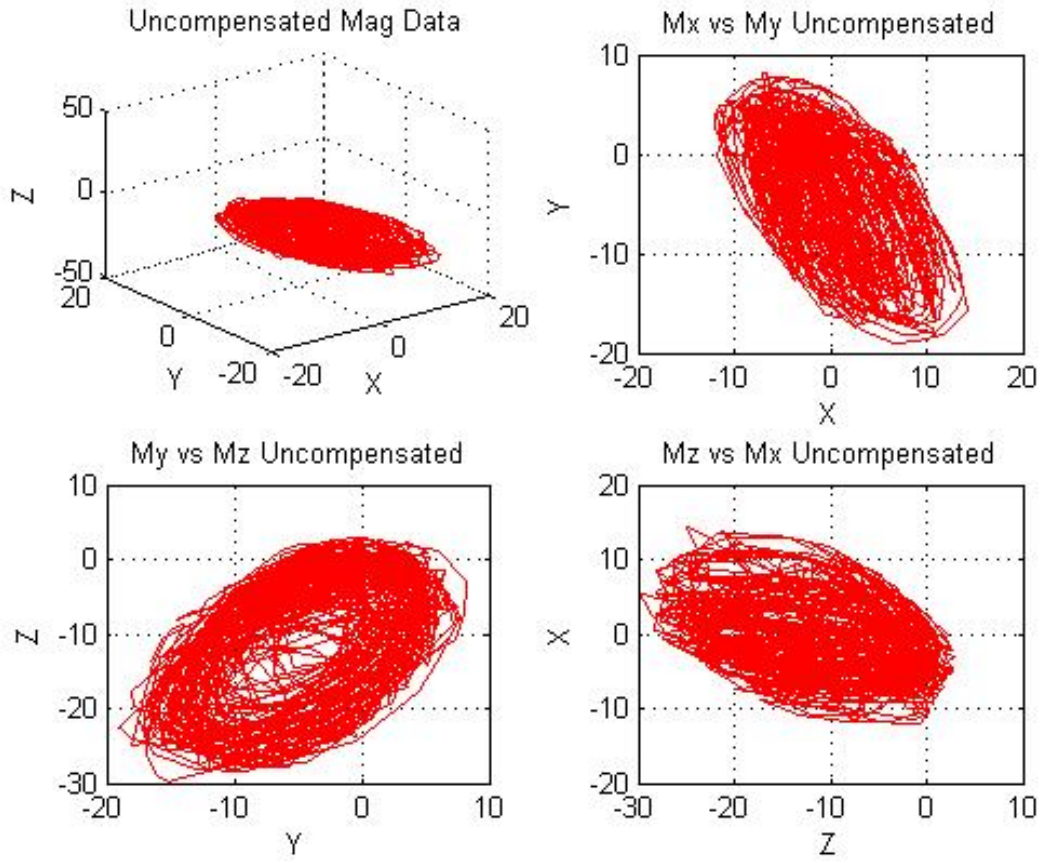


Figura 9 – Medições de um magnetômetro não calibrado. Repare que do centro das medidas não estão em 0 e na forma distorcida [FONTE: ECE Montana]

Dessa forma, é possível verificar a existência dos dois problemas que a calibração visa solucionar: centralizar os círculos (esses erros de *offset* são denominados *hard-iron*) e deixá-los com o mesmo raio, e, portanto, deixando os eixos com a mesma sensibilidade (solucionando os erros denominados de *soft-iron*).

É necessário, também, entender a natureza da causa de cada erro. As distorções *hard-iron* são provocadas por materiais que causam uma distorção de caráter aditivo no campo magnético local (KONVALIN, 2009). Ou seja, pode ser causado por componentes eletrônicos energizados (caixas de som, indutores, eletroímãs, etc) ou pela presença de componentes ferromagnéticos nas proximidades do sensor (como uma carcaça de ferro). Dessa forma, para resolver esse problema é necessário que o *hardware* esteja já montado em sua configuração final, rotacionar o sensor por todas as posições possíveis, e, então, algebricamente, adicionar ou subtrair valores de forma que os círculos fiquem centrados todos em zero.

Por fim, as distorções de *soft-iron* apresentam um caráter um pouco diferente. Elas

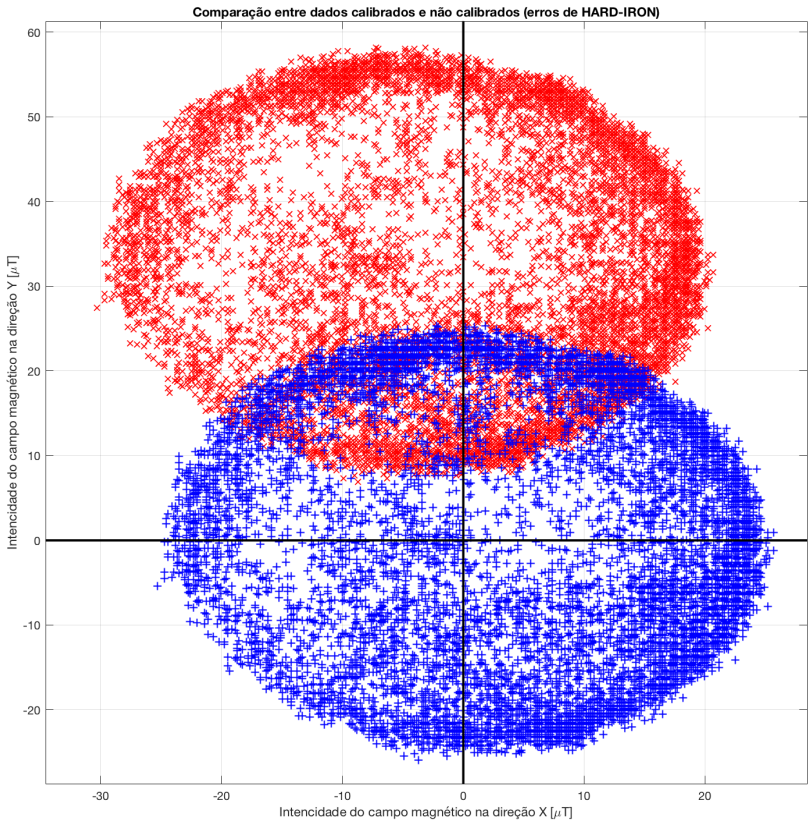


Figura 10 – Medidas aferidas no sensor MPU9250 com e sem calibração. Elipse azul mostra os valores do campo magnético corrigidos pelo erro de *Hard-Iron* e a vermelha mostra sem calibração.

são causadas devido à montagem do sensor propriamente dito, que depende do tipo de material utilizado até a posição do módulo em relação ao campo magnético da terra. A figura 10 ilustra um comportamento típico de um sensor sofrendo esse efeito.

Para se resolver isso, a abordagem é transformar algebricamente essa elipse em um círculo. Para isso, se aplica uma matriz de rotação, para que os eixos da elipse passem a coincidir com os eixos de referência, calcula-se um fator de escala que é a razão do eixo principal em relação ao eixo secundário da elipse, e então se multiplica os valores do eixo secundário por esse fator. Por fim, aplica uma matriz de rotação inversa à primeira para que a referência das leituras passe a ser a mesma do sensor. Aplica-se esse procedimento para todos os três eixos, dois a dois.

Essa abordagem apresentada é mais complexa matematicamente e exige um esforço computacional bem maior que o dos *hard-iron* erros.

Após aplicadas essas correções, devemos ter uma situação semelhante a figura a

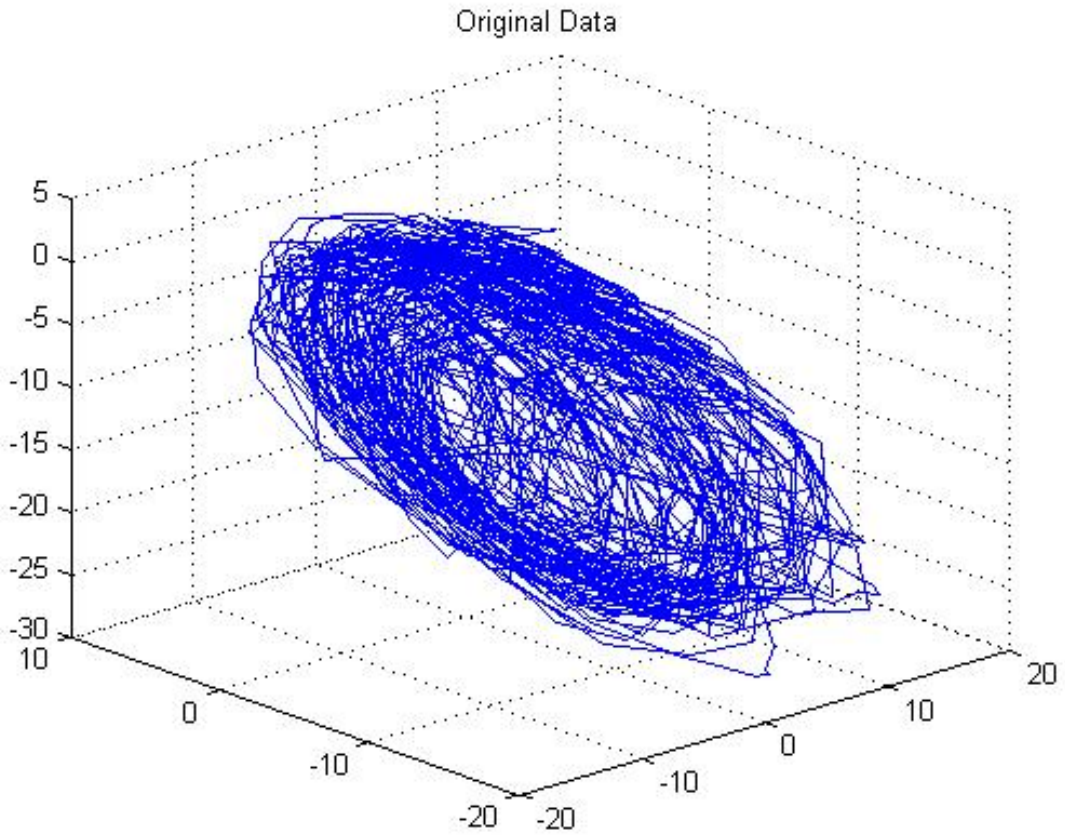


Figura 11 – *Mx* vs *My* vs *Mz* de um magnetômetro com as distorções *soft-iron* não corrigidas em azul, e corrigidas em vermelho.

seguir.

1.6 Filtro de Madgwick

Sebastian O.H. Madgwick escreveu em 2010 o que seria a principal ferramenta desse trabalho. O filtro desenvolvido por Madgwick é específico para ser usado com sensores IMU e garantiu em seus ensaios um erro dinâmico inferior a 1.2 graus (Tabela 1.6). Esse é um filtro estimativo, ou seja, são necessárias algumas iterações para que o resultado convirja para a estimativa ótima.

O Filtro de Madgwick é otimizado para sensores de baixo custo onde o ruído e componentes DC aditivas e variantes no tempo estão presentes nas leituras e tendem a não permitir um posicionamento absoluto decente. O filtro mostrou-se robusto mesmo a baixas taxas de amostragem (fig. 1.6) e tão capaz quanto o conhecido filtro de Kalman, porém exigindo menos poder computacional (MADGWICK, 2010).

Integrando as velocidades angulares medidas por um giroscópio no tempo é pos-

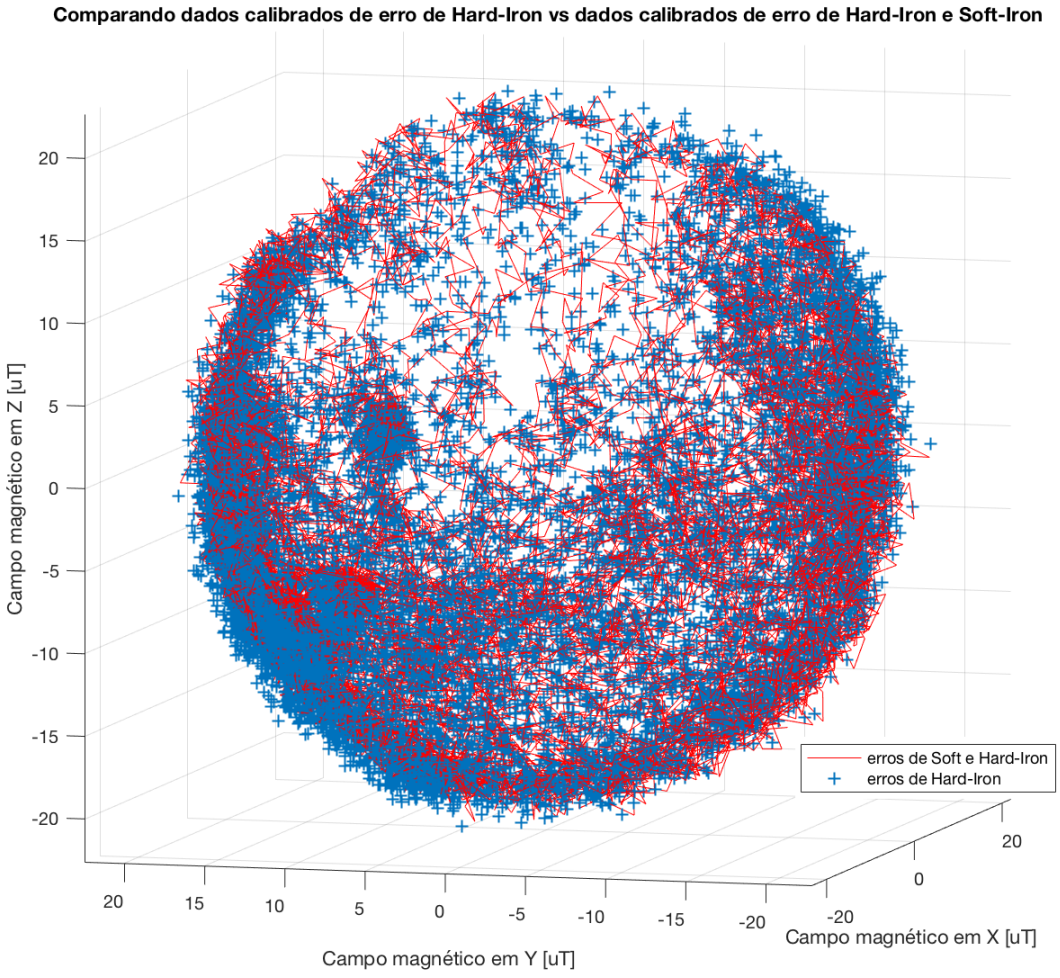


Figura 12 – *Mx*, *My* e *Mz* de um magnetômetro calibrado [FONTE: ECE Montana]

sível, se as condições iniciais são conhecidas, obter a orientação no espaço de um sensor. Entretanto, a integração de pequenos erros e ruídos irá levar ao um acúmulo de erros no posicionamento calculado. Um magnetômetro bem calibrado e um acelerômetro irão medir o campo magnético da terra e sua gravidade, respectivamente, e, assim, fornecer um posicionamento absoluto. Porém estes também são suscetíveis a ruídos. Acelerações provenientes do movimento do sensor irão invalidar a medida da direção da gravidade, por exemplo. O objetivo do filtro de Madgwick é computar uma única estimativa de posição a partir da fusão de mensurações do giroscópio, acelerômetro e magnetômetro. A figura 12 mostra a performance dos filtros de Kalman e Madgewick em função da taxa de amostragem.

Erros nos ângulos de Euler	**Algoritmo - filtro de Kalman**	**Filtro de Madgwick**
RMS[ϕ_ϵ] estático	0.789º	0.581º
RMS[ϕ_ϵ] dinâmico	0.769º	0.625º
RMS[θ_ϵ] estático	0.819º	0.502º
RMS[θ_ϵ] dinâmico	0.847º	0.668º
RMS[ψ_ϵ] estático	1.150º	1.073º
RMS[ψ_ϵ] dinâmico	1.344º	1.110º

Tabela 2 – Erros RMS estático e dinâmico do filtro baseado em Kalman e o filtro proposto por Madgwick

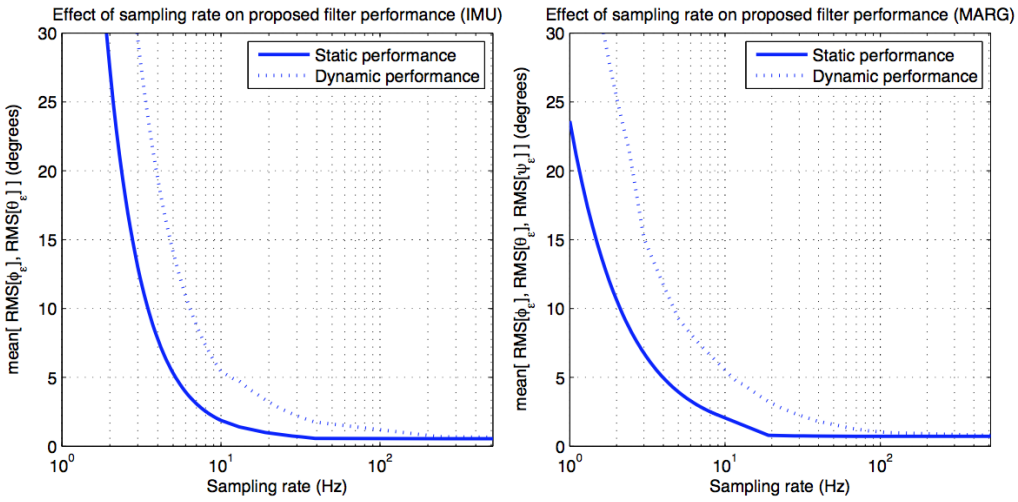


Figura 13 – Efeito da taxa de amostragem na performance do filtro (MADGWICK, 2010)

1.7 *Quaternion*

A palavra *Quaternion* significa "conjunto de quatro". Na matemática, *quaternions* denotam uma uma representação no espaço $\mathbb{R}^3$ e sua álgebra facilita o cálculo de rotações. Quem desenvolveu essa álgebra foi HAMILTON, em 1844, quando ele estudava métodos para calcular multiplicações em vetores com três dimensões. Hamilton trouxe ao mundo a seguinte equação que é a base para a matemática dos *quaternions*. Sendo i, j e k os vetores unitários representando os três eixos ortogonais, a relação identidade dos quaternions é dada por:

$$\hat{\boldsymbol{i}}^2 = \hat{\boldsymbol{j}}^2 = \hat{\boldsymbol{k}}^2 = \hat{\boldsymbol{i}}\hat{\boldsymbol{j}}\hat{\boldsymbol{k}} = -1 \tag{1.1}$$

Como se deve ter percebido, há uma relação da álgebra dos *quaternions* com a álgebra complexa. Um *quaternion* é definido pela soma de um escalar q_0 e um vetor

$\mathbf{q} = (q_1, q_2, q_3)$, ou seja:

$$\hat{\boldsymbol{q}} = q_0 + \boldsymbol{q} = q_0 + q_1\hat{\boldsymbol{i}} + q_2\hat{\boldsymbol{j}} + q_3\hat{\boldsymbol{k}} \tag{1.2}$$

ou

$$\hat{\boldsymbol{q}} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix} \tag{1.3}$$

.

O conjugado de um *quaternion* é dado por:

$$\hat{\boldsymbol{q}}^* = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \end{bmatrix} \tag{1.4}$$

.

E exitem duas operações possíveis: adição e multiplicação; onde a primeira é comutativa e a segunda não necessariamente, logo $\hat{\boldsymbol{p}}\hat{\boldsymbol{q}} \neq \hat{\boldsymbol{q}}\hat{\boldsymbol{p}}$.

1.7.1 Adição e Multiplicação

A adição de *quaternions* segue a equação

$$\hat{\boldsymbol{p}} + \hat{\boldsymbol{q}} = (p_0 + q_0) + (\boldsymbol{p} + \boldsymbol{q}) = (p_0 + q_0) + (p_1 + q_1)\hat{\boldsymbol{i}} + (p_2 + q_2)\hat{\boldsymbol{j}} + (p_3 + q_3)\hat{\boldsymbol{k}} \tag{1.5}$$

E a multiplicação de dois *quaternions* é representada pela matriz

$$\hat{\boldsymbol{p}} \cdot \hat{\boldsymbol{q}} = \begin{bmatrix} p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3 \\ p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2 \\ p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1 \\ p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0 \end{bmatrix}^T \tag{1.6}$$

1.7.2 Rotação

Um *quaternion* unitário define uma rotação, logo, antes de qualquer cálculo usando *quaternions* para representar uma nova orientação rotacionada deve-se primeiro normalizalo.

Outra característica de um *quaternion* é que este é construído por "metade de ângulo", ou seja é construído da seguinte forma:

$$\prescript{A}{B}{\hat{\boldsymbol{q}}} = \begin{bmatrix} cos(\frac{\theta}{2}) & -r_x sin(\frac{\theta}{2}) & -r_y sin(\frac{\theta}{2}) & -r_z sin(\frac{\theta}{2}) \end{bmatrix} \tag{1.7}$$

Assim, esse *quaternion* representa a orientação do quadro B relativo ao quadro A e ${}^{A}\hat{\boldsymbol{r}}$ é um vetor definido em A.

Um vetor definido em B (${}^{B}\boldsymbol{v}$) é calculado a partir do vetor definido em A (${}^{A}\boldsymbol{v}$) por meio de uma rotação anti-horária no plano 3D e no plano ortogonal por $\frac{\theta}{2}$ (${}^{A}_{B}\hat{\boldsymbol{q}}$); seguida por uma rotação no sentido anti-horário no plano 3D e horário no plano ortogonal por $\frac{\theta}{2}$ (${}^{A}_{B}\hat{\boldsymbol{q}}^{*}$), ou seja:

$$ {}^{B}\boldsymbol{v} = {}^{A}_{B}\hat{\boldsymbol{q}} \cdot {}^{A}\boldsymbol{v} \cdot {}^{A}_{B}\hat{\boldsymbol{q}}^{*} \tag{1.8} $$

onde

$$ \boldsymbol{v} = \begin{bmatrix} 0 & v_1 & v_2 & v_3 \end{bmatrix} \tag{1.9} $$

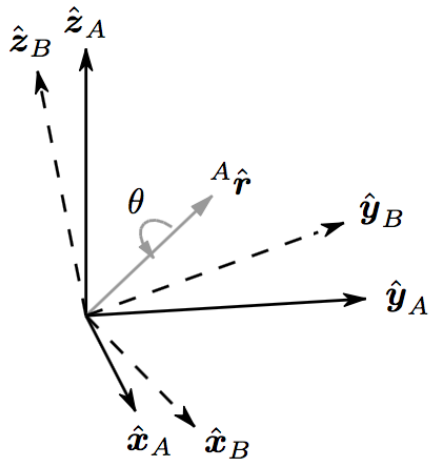


Figura 14 – A orientação do quadro B é alcançada a partir de uma rotação, alinhada com o quadro A, de ângulo θ ao redor do eixo ${}^{A}\hat{\boldsymbol{r}}$

A orientação ${}^{A}_{B}\hat{\boldsymbol{q}}$ pode ser representada pela matriz rotacional ${}^{A}_{B}\boldsymbol{R}$ definida pela equação (1.10).

$$ {}^{A}_{B}\boldsymbol{R} = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & 2q_0^2 - 1 + 2q_2^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & 2q_0^2 - 1 + 2q_3^2 \end{bmatrix} \tag{1.10} $$

1.8 Software

Para a simulação em software, foi utilizada a *engine* de jogos Unity3D (versão 2017.2.0f3) desenvolvida pela *Unity Technologies*, que conta com uma série de ferramen-

tas para a reconstrução do episódio do acidente, dentre elas, a aplicação de fenômenos físicos(gravidade, resistência do ar, torques, colisões, atrito) e, por ela ser uma ferramenta muito versátil, é possível se produzir um ambiente de simulação bastante intuitivo e fiel às medições obtidas. Essa plataforma foi a escolhida por ser a plataforma mais amplamente utilizada para o desenvolvimento de jogos cuja licença, para nosso caso, é gratuita. Vários estúdios, de grande e pequeno porte, utilizam dessa ferramenta para a sua produção. Alguns títulos produzidos nela são como *Life is Strange* da *Square Enix*, *CupHead* da *Studio MHDR* e *Temple Run* da *Imangi Studios*.

Parte II

Desenvolvimento Hardware

2 Desenvolvimento do Hardware

2.1 Introdução

A seguir, temos uma imagem do hardware final do nosso protótipo.

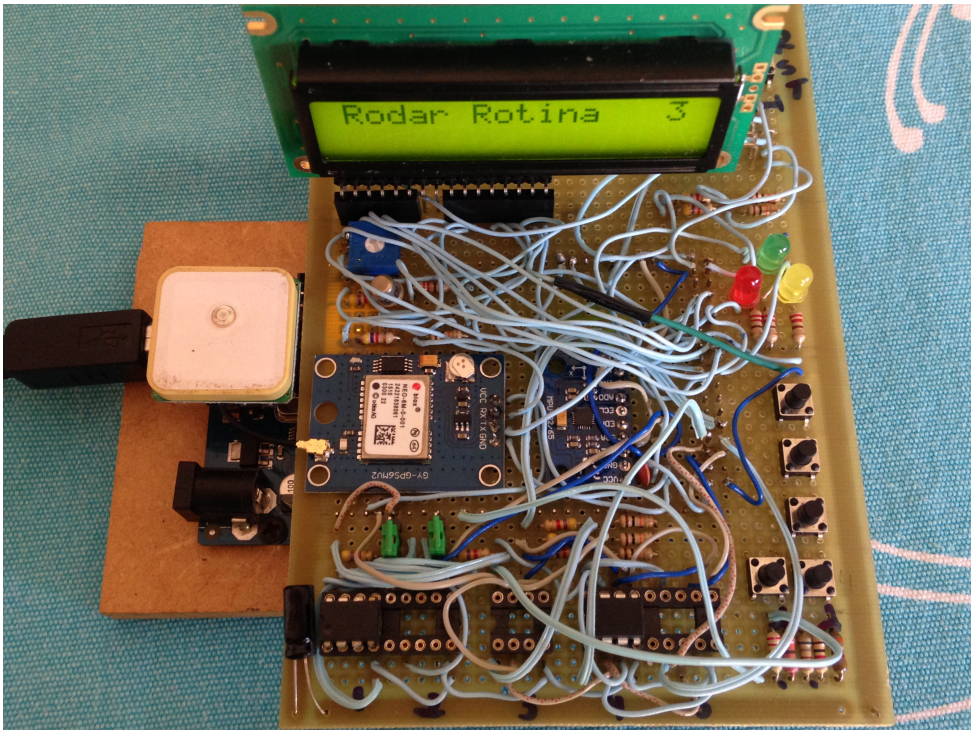


Figura 15 – Foto da plataforma inercial “Caixa-preta”

A plataforma de coleta de dados foi concebida de modo a facilitar a interface com o usuário, acelerando dessa forma o desenvolvimento do *firmware* e suas funcionalidades. Para tanto, dispõe dos seguintes dispositivos de *feedback* e controle:

- *display LCD* 16x2;
- três LEDs (vermelho, amarelo e verde);
- cinco chaves sem retenção (*push-button*).

Informações importantes para o uso da plataforma são disponibilizadas no *display* e os modos de uso podem ser escolhidas usando-se as chaves. São cinco operações possíveis, dispostas em um *menu* vertical, numerado de 1 a 5, são eles:

1. Testar MARG;

2. Testar GPS;
3. Rodar Rotina;
4. Salvar Dados;
5. Apagar Dados.

No modo "Testar MARG" o dispositivo lê os sensores (acelerômetro, giroscópio e magnetômetro) e envia os dados na porta serial (USB) do Arduino Mega da seguinte sequencia: "Raw:ax,ay,az,gx,gy,gz,mx,my,mz". Esse modo serve para testar se os dados estão coerentes e também para realizar a calibração do magnetômetro. Vale ressaltar que os dados estão em formato inteiro, ou seja, não representa valores absolutos. Deve se aplicar a escala no qual o sensor está ajustado para que tenha-se valores reais.

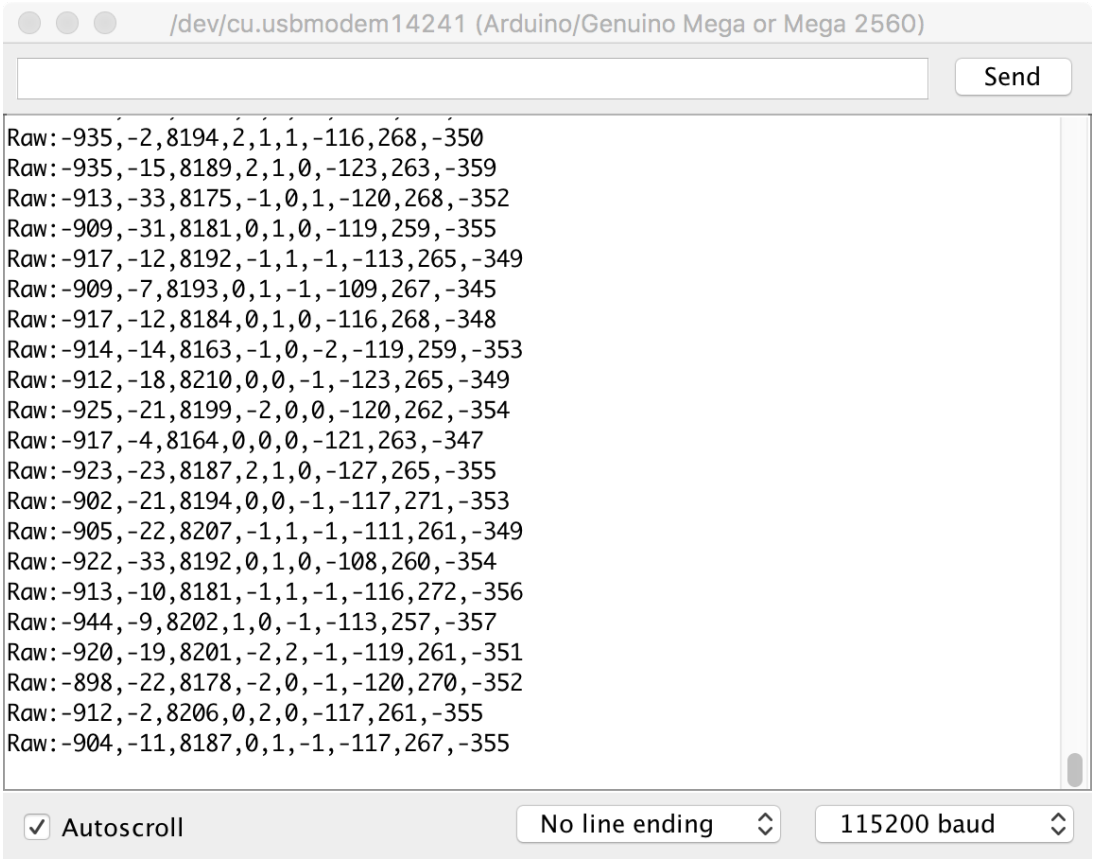


Figura 16 – Dados crus sendo externados na porta serial

"Testar GPS" tem como objetivo testar a recepção dos sinais de GPS. Nesse modo, a plataforma indica, pelo *display*, qual é o tipo de "GPSfix" que receptor está conseguindo naquele momento. Esse dado é uma *flag* importante que indica se informações como velocidade e/ou posição tem validade ou não.

O terceiro modo, "Rodar Rotina", é a rotina de funcionamento da plataforma. Nesse modo a plataforma age como uma caixa-preta, ou seja, fica em seu *loop* coletando,

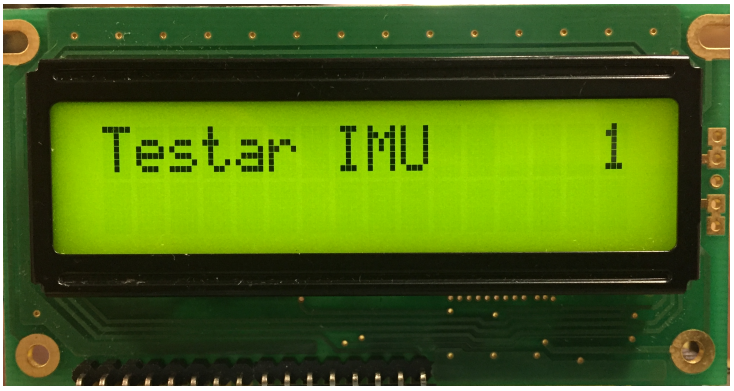


Figura 17 – imagem mostrando o *display* no modo Testar IMU

Figura 18 – imagem mostrando o *display* no modo Testar GPS

Figura 19 – imagem mostrando o *display* no modo Rodar Rotina

salvando e descartando dados em modo sequencial até que um evento de interrupção (batida) seja identificado. Quando um evento ocorre a plataforma coleta mais meio período de dados e então os transfere para a EEPROM. Em resumo, metade da memória retém os dados antes do acidente e a outra metade os dados do pós acidente.

"Salvar Dados" não salva exatamente os dados, mas transfere os dados da EEPROM para a porta serial de modo que possa ser salvo por um computador. Como não há uma memoria removível na plataforma, essa é a única maneira de acessar os dados

salvos na EEPROM.

O último modo faz exatamente o que seu nome diz, apaga os dados da EEPROM.

2.2 Coleta de Dados

Os dados do sensor MARG são extraídos usando um *driver* disponibilizado pela fabricante de *Open Hardwares* Sparkfun. Esse programa foi modificado pontualmente para que essa leitura fosse feita de uma só vez, em vez de precisar de uma função para cada um dos três sensores incluídos no MARG. Esse *driver* também serve para configurar os sensores e calibrá-los (giroscópio e acelerômetro).

Os dados do GPS são extraídos de maneira mais fácil. O u-blox NEO-6m permite que uma **mensagem** de informação seja requisitada ou que ela seja enviada periodicamente na porta serial. Uma mensagem de informação é um conjunto de dados formatados em um protocolo proprietário da u-blox, chamado UBX, que são enviados ao computador ou microcontrolador. Suas características são:

- Compacta - usa dados binários de 8 *bits*;
- protegido por *Checksum*;
- modular - usa uma identificação de mensagem de dois estágios.

A estrutura do pacote de informação UBX é representado na figura 17. Em seguida, listamos os elementos básicos desse vetor de informações.

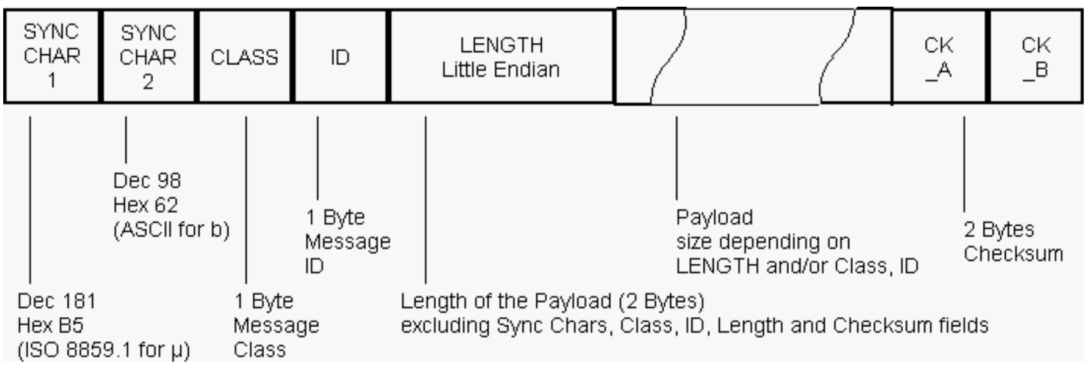


Figura 20 – Estrutura da mensagem UBX (UBLOX).

- Toda mensagem começa com dois *bytes*: 0xB5 0x62;
- segue então um *byte* de Classe (Class). A Classe define o conjunto básico da informação;

- em seguida, um *byte* de identificação (ID) que define a mensagem que será entregue;
- então dois *bytes* informam o tamanho da informação que vem em seguida (*LENGTH*);
- o *Payload* é a informação desejada;
- por fim, dois *bytes* de checksum são inclusos.

O *checksum* usa do algoritmo de 8 *bits* de Fletcher, usado no padrão TCP (RFC1145). Ele pode ser calculado como ilustrado na figura 18, onde *buffer*[N] contém os dados onde devem ser calculados o *checksum* e CK_A e CK_B são do tipo 8-*bit unsigned integer*.

```
CK_A = 0, CK_B = 0
For(I=0;I<N;I++)
{
    CK_A = CK_A + Buffer[I]
    CK_B = CK_B + CK_A
}
```

Figura 21 – Algoritmo de cálculo do *checksum* (UBLOX).

Assim, toda vez que deseja-se coletar dados é necessário apenas enviar um comando, uma sequência de *bytes*, para o módulo. Feito um pedido, o receptor responde com um conjunto de *bytes* organizados no padrão UBX e, ao final, deve-se calcular o *checksum* para garantir a integridade dos dados.

Há diversas mensagens possíveis para extrair dados como posição e velocidade. Escolhemos a mensagem NAV-SOL que tem exatamente 60 *bytes*, dos quais 52 são de informações úteis, sendo que apenas 40 serão usadas nesse projeto. São eles:

1. iTOW (tempo GPS em milissegundos iniciando na primeira hora da semana;
2. week (número de semanas desde 6 de janeiro de 1980);
3. ecefX (posição no eixo X em coordenadas ECEF em cm);
4. ecefY (posição no eixo Y em coordenadas ECEF em cm);
5. ecefZ (posição no eixo Z em coordenadas ECEF em cm);
6. pAcc (estimativa de acurácia da posição 3D em cm);
7. ecefVX (velocidade no eixo X em coordenadas ECEF em cm/s);
8. ecefVY (velocidade no eixo Y em coordenadas ECEF em cm/s);
9. ecefVZ (velocidade no eixo Z em coordenadas ECEF em cm/s);

10. sAcc (estimativa de acurácia da rapidez em cm/s);
11. pDOP (DOP da posição).

Para saber se esses dados estão válidos (são mesmo o que dizem que são) é preciso verificar o *byte* "flag" da mensagem onde cada um dos quatro *bit* menos significativos indica a validade de um dos seguintes itens:

- *bit* 0: GPSfixOK (dados dentro do mínimo DOP e ACC);
- *bit* 1: se 1, DGPS usado;
- *bit* 2: se 1, número da semana válido;
- *bit* 3: se 1, TOW válido;

2.3 Rotina e Armazenamento

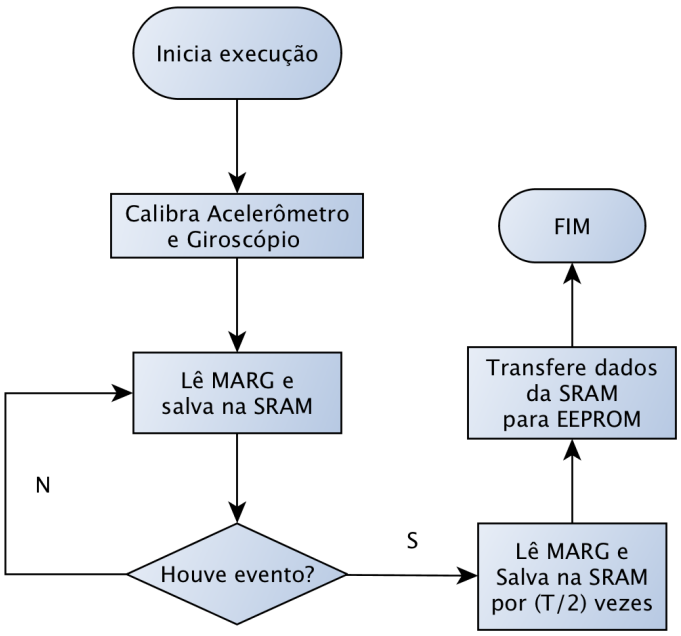


Figura 22 – Diagrama do funcionamento da rotina de coleta de dados do MARG

O rotina principal da plataforma caixa-preta é coletar dados do MARG continuamente, armazenando-os na SRAM, em um ciclo perpétuo, até que seja observado um evento (batida) que gera o desencadeamento de uma subrotina que completa a coleta de dados por meio período e, então, salva os dados na EEPROM. A coleta dos dados do MARG é feita a cada dez milissegundos, e a cada 100 ms são coletados os dados do GPS, armazenando esses na memória do microcontrolador.

Essas taxas praticadas são as taxas compatíveis com os sensores. O acelerômetro e giroscópio até são capazes de exceder esses números, podendo ter uma taxa de 1000 Hz, porém, como o magnetômetro é limitado a 100 Hz preferimos manter essa taxa para todos. O GPS pode operar até 10 Hz, logo, estamos operando-o na capacidade máxima.

Foi desenhada a seguinte sequência de ações (ilustrada na Figura 19, simplificadamente):

1. Coleta-se e salva-se 32 amostras do MARG, totalizando 576 *bytes*, na RAM do microcontrolador;
2. a cada 10 amostras do MARG, coleta-se uma amostra do GPS e a salva em um *buffer* circular com capacidade para 8 amostras de 40 *bytes*;
3. coletada a trigésima segunda amostra, envia esse vetor de dados para a SRAM;
4. volta ao item 1.

Quando se tem uma batida, guarda-se o endereço da memória onde foi a última escrita, armazena-se mais 3616 amostras do MARG e nenhuma mais do GPS. Por fim, todo o conteúdo da memória SRAM é transferido para a EEPROM, começando do endereço calculado a partir do último endereço antes da batida menos 65.664. Se o endereço resultante for negativo, soma-se 0x1FFFF (131.071).

Os números acima não foram pensados ao acaso. As duas memórias, têm, exatamente 131072 *bytes*. Fazendo-se 114 escritas na SRAM antes do evento, 113 após, garante 7264 amostras aferições inerciais e magnéticas, resultando em 130752 *bytes*, faltando assim 320 que serão preenchidos com as 8 amostras do GPS. Faz-se, então, um uso completo do armazenamento disponível.

Parte III

Desenvolvimento Software

3 Desenvolvimento do Software

3.1 Plataforma Unity3D

O Unity3D é uma plataforma utilizada para o desenvolvimento de jogos criada pela Unity Technologies, tendo sua forma principal de desenvolvimento a interface gráfica, mostrada na figura abaixo.

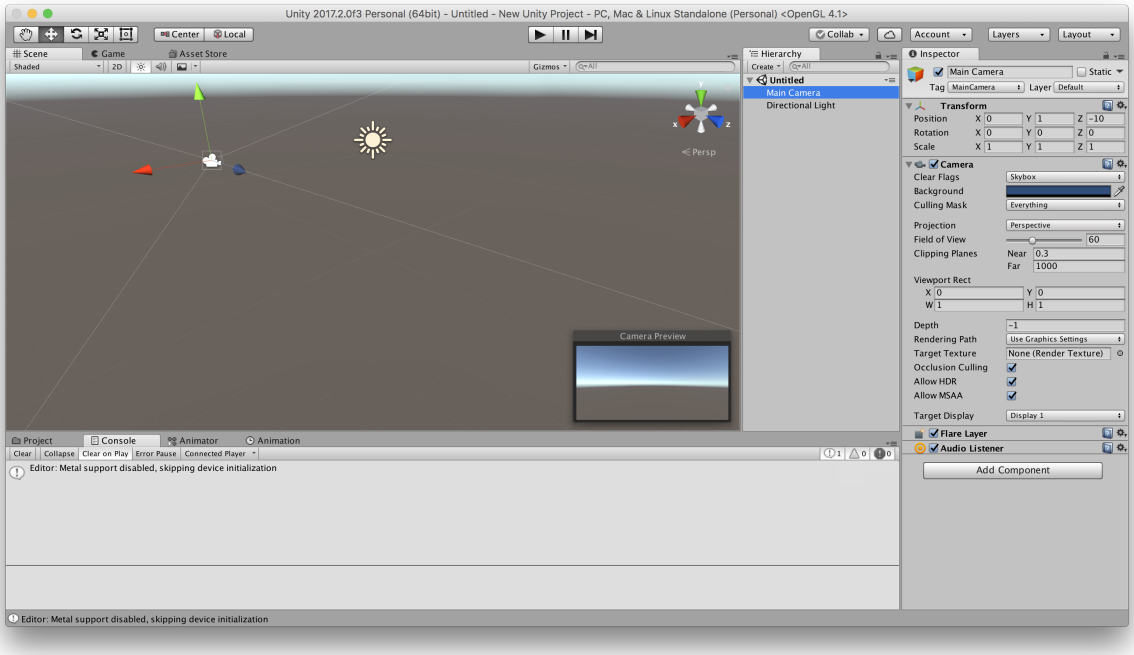


Figura 23 – Tela de edição de projeto do Unity3D

A ideia de utilização dessa plataforma é criar um jogo em que os comandos de movimento sejam fornecidos pela plataforma "caixa-preta" previamente montada. Assim, serão utilizados dados de aceleração nos três eixos, de rotação por meio do IMU e dados de velocidade de movimento, fornecido pelo GPS. Esses dados serão utilizados em formato de arquivo de texto, que será tratado de forma computacional no ambiente de desenvolvimento. Todos os códigos foram feitos na linguagem C#. Atualmente, ela é a plataforma de produção de jogos mais utilizada.

3.2 Licença

Esse software conta com três tipos de licença de utilização, a *Personal*, *Plus* e *Pro*, sendo as duas últimas pagas. Para a utilização da *Personal*, o jogo deve ter uma

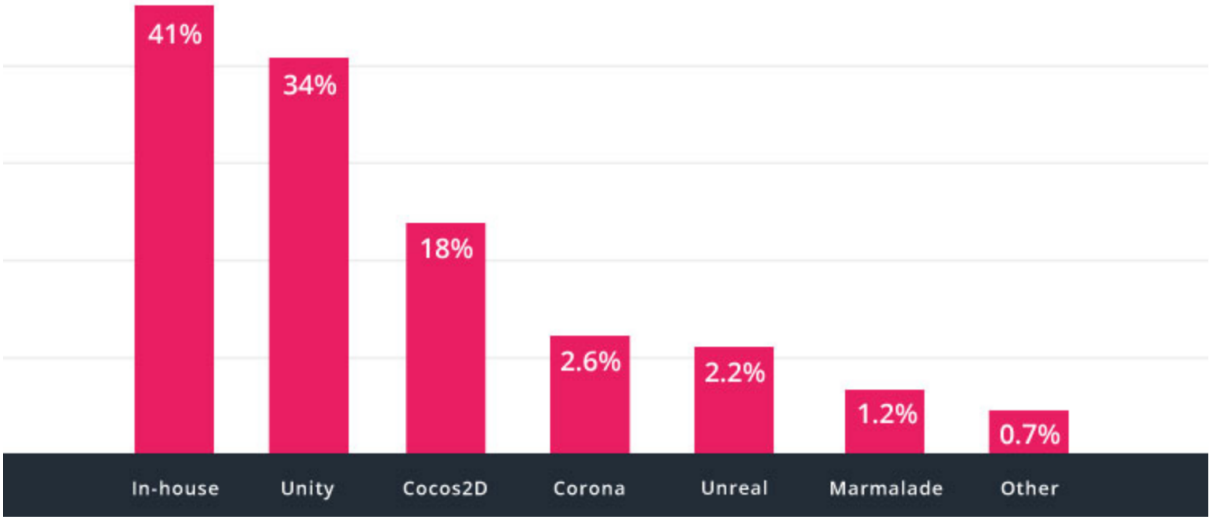


Figura 24 – Gráfico comparativo dos percentuais da presença das IDEs de produção de jogos em jogos mobile gratuitos

receita menor que 100 mil dólares por ano, escalando para 200 mil na *Plus* e ilimitado na Pro. Assim, esse projeto utilizará da licença *Personal*, já que não será utilizado para fins comerciais, logo não haverá receita.

3.3 Simulador

A seguir temos uma cena com todos os componentes do simulador.

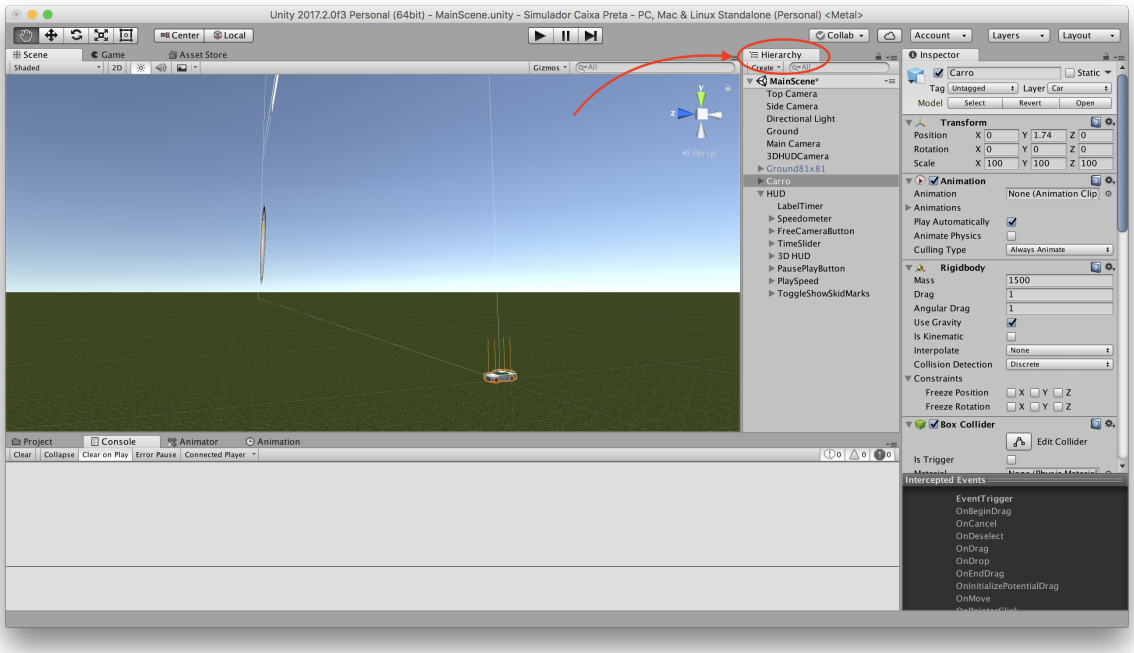


Figura 25 – Cena principal do simulador

Na aba *hierarchy*, em destaque na figura acima, temos todos os *GameObjects* que foram utilizados para a sua construção (câmeras, objetos, colisores, corpos rígidos, etc). Os mais importantes serão detalhados em seguida

3.3.1 GameObject Carro

No carro, temos todos os componentes que fazem a física e descrevem o comportamento do carro. A começar, foi criado um corpo rígido (*RigidBody*). Esse elemento tem a função de introduzir os comportamentos físicos básicos, como massa, gravidade, colisões, torque, atrito, resistência do ar.

Porém, para que as propriedades físicas funcionem adequadamente, é necessário delimitar os volumes ocupados pelo objeto, evitando assim que ele atravesse o chão e para ter uma semelhança maior com a realidade para quando se aplicar as acelerações dadas pelo acelerômetro. Para tal, se utilizam *colliders*, que são formas geométricas que são posicionadas para se dar o formato do objeto. Idealmente deveriam ter sido utilizado vários prismas para tal, mas como isso é um processo que aumentaria muito as exigências computacionais e não é necessária uma forma tão acurada para se ter um resultado próximo do ocorrido. Foram utilizados apenas dois *colliders* retangulares, como mostrado na figura abaixo.

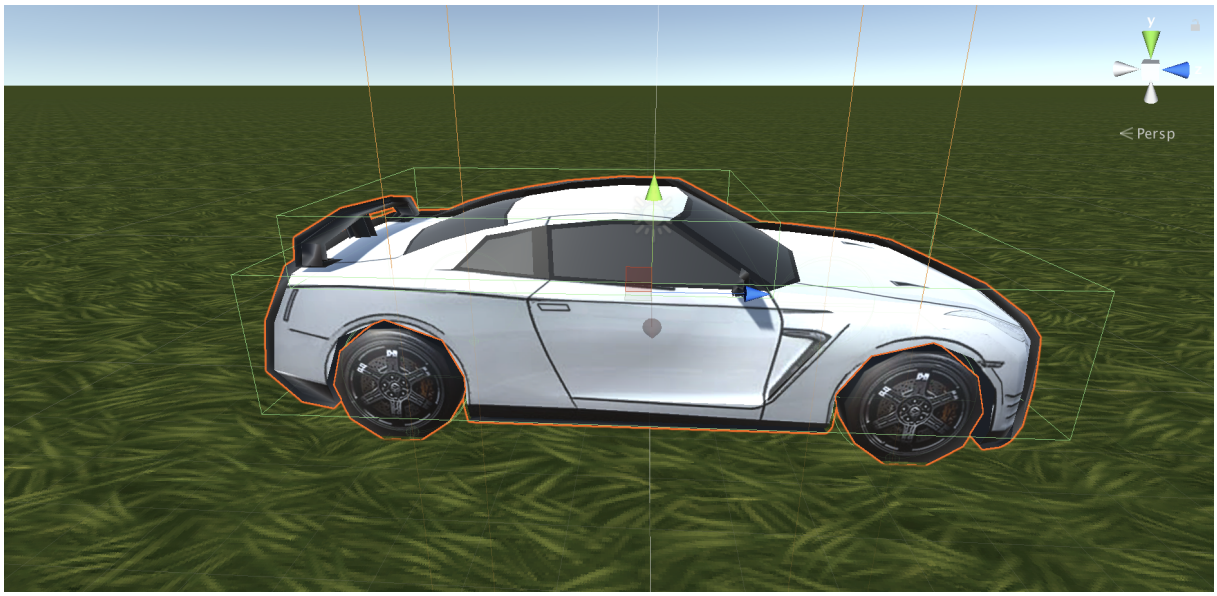


Figura 26 – *Colliders* (caixas verdes) formando o carro

Além desses componentes, foram adicionadas também marcas dos pneus. Porém, essas marcas, ao contrário das marcas que ocorrem na realidade, elas acompanham todo o movimento dos pneus traseiros, estando eles andando em linha reta, na curva ou no ar,

para que dessa forma se possa ter, a qualquer momento, um registro do movimento do carro durante o episódio.

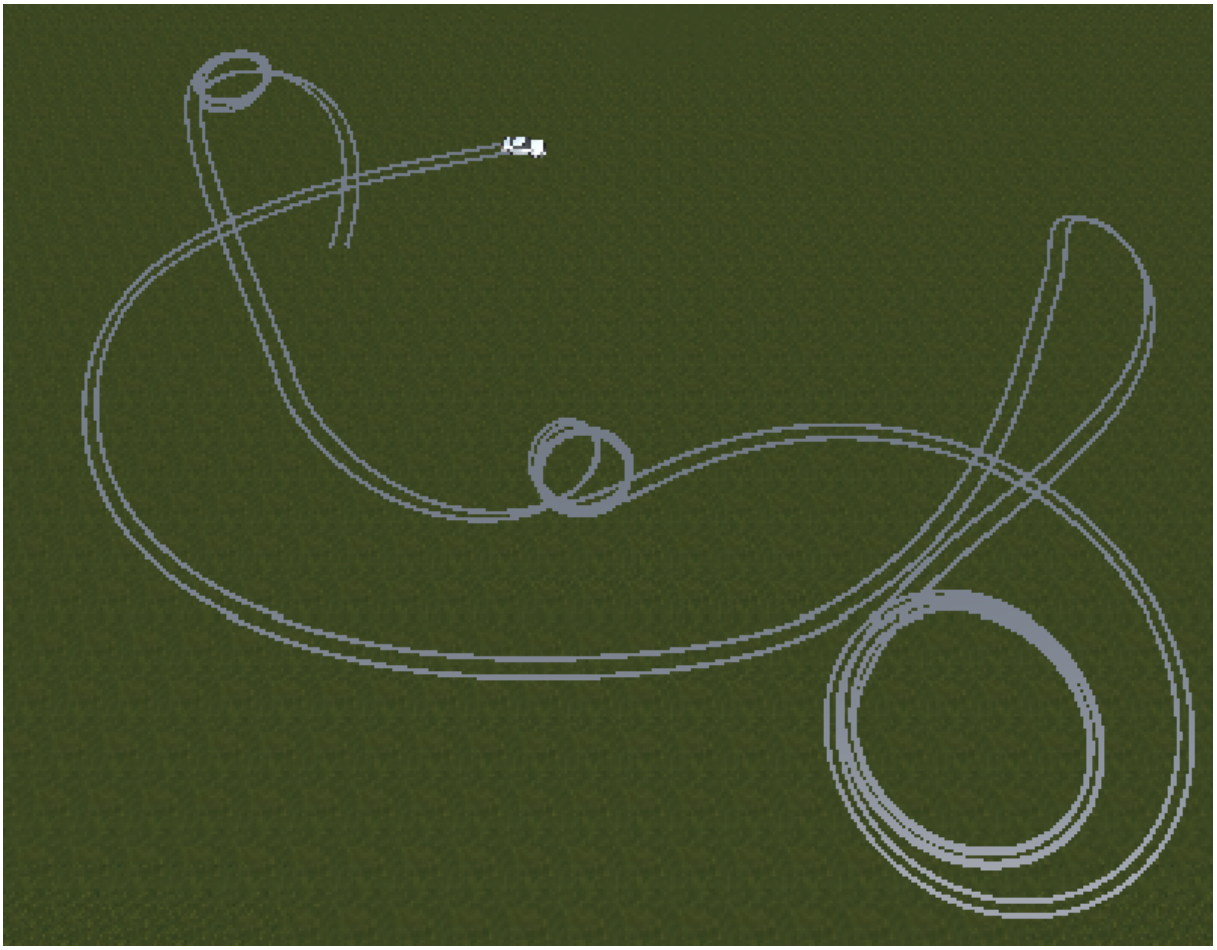


Figura 27 – Exemplo da simulação das marcas dos pneus.

3.3.2 HUD

Na HUD temos os elementos que mostrarão os parâmetros para o usuário. Há um velocímetro que mostrará a magnitude da velocidade do carro, três eixos normalizados que mostram as acelerações que estão atuando no carro em cada determinado instante, temos, também, um botão para desvincular a câmera principal do carro, para que assim se possa ter outro ângulo da cena

Nela, colocamos uma caixa de seleção para poder habilitar ou desabilitar as marcas de pneu, e dessa forma facilitar a visualização do movimento em determinados casos.

3.3.3 Funcionamento

Ao iniciar a simulação, o simulador começa um processo de leitura de dados em uma taxa temporal de 1:10 (tempo real para tempo de simulação). Nesse instante, a HUD

Figura 28 – Tela da simulação com o HUD

é desabilitada, até o fim do carregamento. Durante o carregamento, os *quaternions* e as acelerações recebidas pelo hardware são aplicados no corpo rígido do carro, e logo em seguida, sua posição nos três eixos é salva junto dos seus três ângulos de Euler.

Após terminada a leitura dos dados, o ambiente de simulação está pronto para a reprodução. A HUD passa a ser reabilitada e, assim o usuário pode pausar, alterar a velocidade da reprodução, ou selecionar o momento da simulação que deseja visualizar arrastando a barra de rolagem. É possível também, pelo botão *Free Camera*, desvincular a câmera principal do objeto do carro e posicioná-la da forma que seja mais conveniente. Para isso, usam-se as teclas WASD para a movimentação, *shift* e *control* para mudar a altura, e as setas para ajustar a inclinação.

Figura 29 – Comparativo Marcas de pneu ligadas e desligadas[todo]

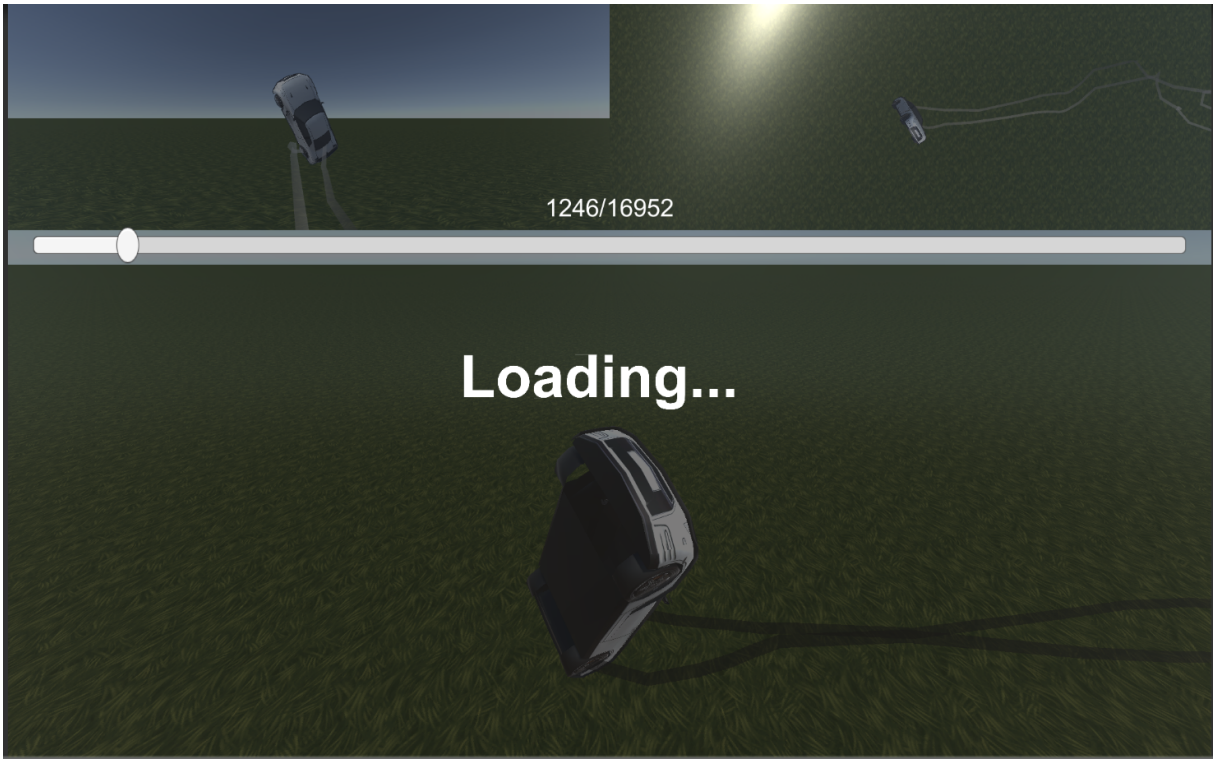


Figura 30 – Tela de carregamento dos dados da simulação

4 Conclusão

Pelo protótipo desenvolvido nesse trabalho, é possível constatar que o uso desse dispositivo traria algum embasamento para a polícia técnica na realização de seus laudos. A caixa-preta age colhendo dados importantes para a reconstrução das forças e atitudes de um veículo automotor em um evento de colisão. Entretanto, de modo a tornar o instrumento mais didático, optou-se por incluir outras funcionalidades que trazem facilidade ao usuário ou desenvolvedor.

Como um TCC, esse desenvolvimento tem objetivos de aprendizado e também de consolidação das teorias e práticas do curso de Engenharia elétrica. Um dos principais avanços nesse trabalho é o uso apropriado do filtro de Madgwick, o qual permite uma reconstrução fidedigna das dinâmicas impostas a um veículo em uma colisão ou qualquer evento que venha a fazer vítimas.

O maior problema do protótipo atual é em questão as suas dimensões. Pelo fato da placa ter sido confeccionada artesanalmente e de seu tamanho, inviabiliza-se boa parte das possibilidades de se testar. Porém, isso poderia ser resolvido utilizando-se de circuitos impressos, já que dessa maneira, as trilhas podem ser feitas nas diferentes camadas da placa, economizando bastante espaço.

Em relação a sua operação, a grande dificuldade está na calibração do magnetômetro. Essa etapa do processo é crucial e deve ser feita sempre que o dispositivo é colocado em uma nova configuração, ou em um novo ambiente, para que se possam ser resolvidas as distorções *hard* e *soft-iron*, e assim, o algoritmo do filtro de Madgewick calcule a orientação correta do veículo.

O simulador desenvolvido é outro avanço importante, pois, disponibiliza para a polícia científica estimativas visuais do ocorrido e embasa seus relatórios com evidências. Entre as melhorias implementadas estão as três vistas diferentes, a capacidade de posicionar a câmera principal de forma livre, as marcas de pneu, o acompanhamento contínuo das forças sofridas, a escala temporal, e a física imposta ao veículo simulado, onde o chão é intransponível e a gravidade é fixa.

Finalmente, em questão ao software, algumas melhorias ou adições poderiam ser feitas. Primeiramente, o tempo de carregamento inicial poderia ser otimizado para que ele fique mais rápido. Segundo, o simulador poderia ler e interpretar os dados do GPS para pegar a velocidade estimada do carro e localizar geograficamente o trajeto do carro, para um melhor entendimento do que aconteceu.

Outra adição poderia ser a leitura dos dados do hardware via comunicação serial,

para que o software pudesse reproduzir em tempo real o que está acontecendo com o módulo, dessa forma podendo testar a confiabilidade da reprodução e a qualidade da calibração do magnetômetro.

Referências

ATMEL. *8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash*. [S.l.]. Disponível em: <http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf>. Citado na página 27.

BASICS, C. Basics of uart communication. Disponível em: <http://www.circuitbasics.com/basics-uart-communication/>. Citado na página 29.

DCD. Diário da câmara do deputados. 2011. Disponível em: <http://imagem.camara.gov.br/Imagem/d/pdf/DCD11JUN2011.pdf>. Citado na página 21.

EPSON. Gyro sensors - how they work and what's ahead. Disponível em: <http://www5.epsondevice.com/en/information/technical_info/gyro/>. Citado na página 26.

ESCHOOLTODAY. The coriollis effect. Disponível em: <http://www.eschooltoday.com/winds/the-coriolis-effect.html>. Citado 2 vezes nas páginas 11 e 26.

G1-PARAíBA. Técnicos realizam perícia em local de acidente que vitimou advogado na pb. 2015. Disponível em: <http://g1.globo.com/pb/paraiba/noticia/2015/01/tecnicos-realizam-pericia-em-local-de-acidente-que-vitimou-advogado-na-pb.html>. Citado 2 vezes nas páginas 11 e 21.

GRIFFIN, D. How does the global positioning system work ? 2011. Disponível em: <http://www.pocketgpsworld.com/howgpsworks.php>. Citado na página 27.

HAMILTON, W. R. On quaternions; or on a new system of imagniaries in algebra. 1844. Disponível em: <https://www.emis.de/classics/Hamilton/OnQuat.pdf>. Citado na página 36.

KONVALIN, C. Compentating for tilt hard iron and soft iron effects. 2009. Disponível em: <https://www.sensorsmag.com/components/compensating-for-tilt-hard-iron-and-soft-iron-effects>. Citado na página 32.

MABE, V. How does a magnetometer work. 2017. Disponível em: <http://sciencing.com/a-magnetometer-work-4913575.html>. Citado na página 26.

MADGWICK, S. O. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. 2010. Disponível em: <http://x-io.co.uk/res/doc/madgwick_internal_report.pdf>. Citado 3 vezes nas páginas 11, 34 e 36.

MAI, T. Global positioning system history. 2012. Disponível em: <https://www.nasa.gov/directorates/heo/scan/communications/policy/GPS_History.html>. Citado na página 27.

MICROCHIP. datasheet eeprom 24fc1025. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/21941B.pdf>. Citado na página 30.

MICROCHIP. Datasheet sram 23lc1025. Disponível em: <http://ww1.microchip.com/downloads/en/DeviceDoc/20005142C.pdf>. Citado 2 vezes nas páginas 11 e 31.

NHTSA. Event data recorder. 2011. Disponível em: <https://www.nhtsa.gov/research-data/event-data-recorder>. Citado na página 21.

RAFTER, M. V. Decoding what's inside your car blackbox. 2014. Disponível em: <https://www.edmunds.com/car-technology/car-black-box-recorders-capture-crash-data.html>. Citado na página 21.

REIS, V. R. dos. I2c – protocolo de comunicação. 2014. Disponível em: <http://www.arduinobr.com/arduino/i2c-protocolo-de-comunicacao/>. Citado na página 29.

RFC1145. Tcp alternate checksum options. Disponível em: <http://www.ietf.org/rfc/rfc1145.txt>. Citado na página 47.

SACCO, F. Comunicação spi – parte 1. 2014. Disponível em: <https://www.embarcados.com.br/spi-parte-1/>. Citado na página 29.

SENSORWIKI. Accelerometer. Disponível em: <http://www.sensorwiki.org/doku.php/sensors/accelerometer>. Citado na página 25.

SOUZA, F. Arduino mega 2560. 2014. Disponível em: <https://www.embarcados.com.br/arduino-mega-2560/>. Citado 2 vezes nas páginas 11 e 28.

UBLOX. Receiver descrition / including protocol specification. Disponível em: <https://www.u-blox.com/sites/default/files/products/documents/u-blox6_ReceiverDescrProtSpec_%28GPS.G6-SW-10018%29_Public.pdf>. Citado 3 vezes nas páginas 11, 46 e 47.

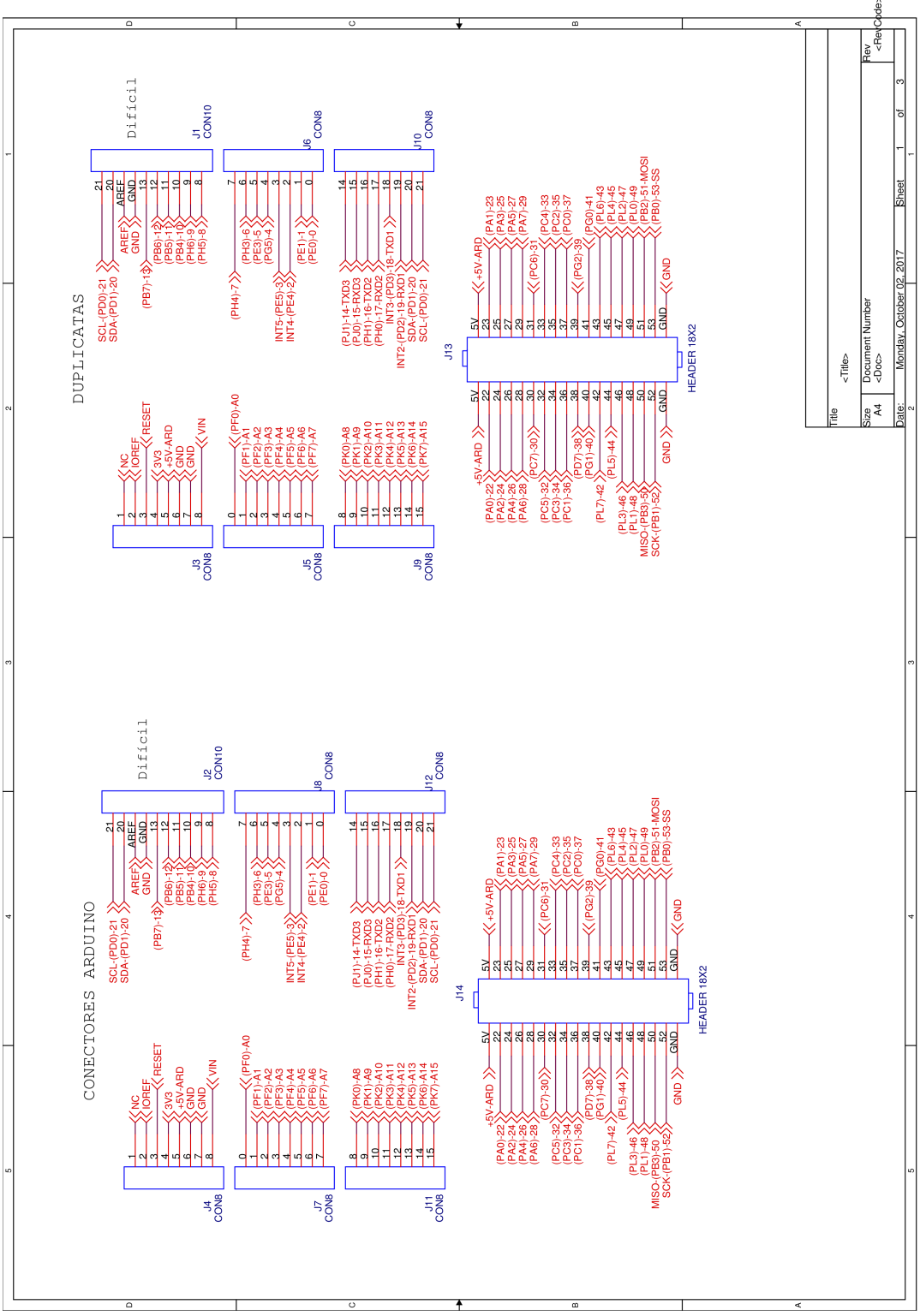
WHO. *Road Traffic Deaths Data by Country*. 2017. Disponível em: <http://apps.who.int/gho/data/node.main.A997>. Citado na página 19.

WINER, K. Simple and effective magnetometer calibration. 2017. Disponível em: <https://github.com/kriswiner/MPU6050/wiki/Simple-and-Effective-Magnetometer-Calibration>. Citado na página 31.

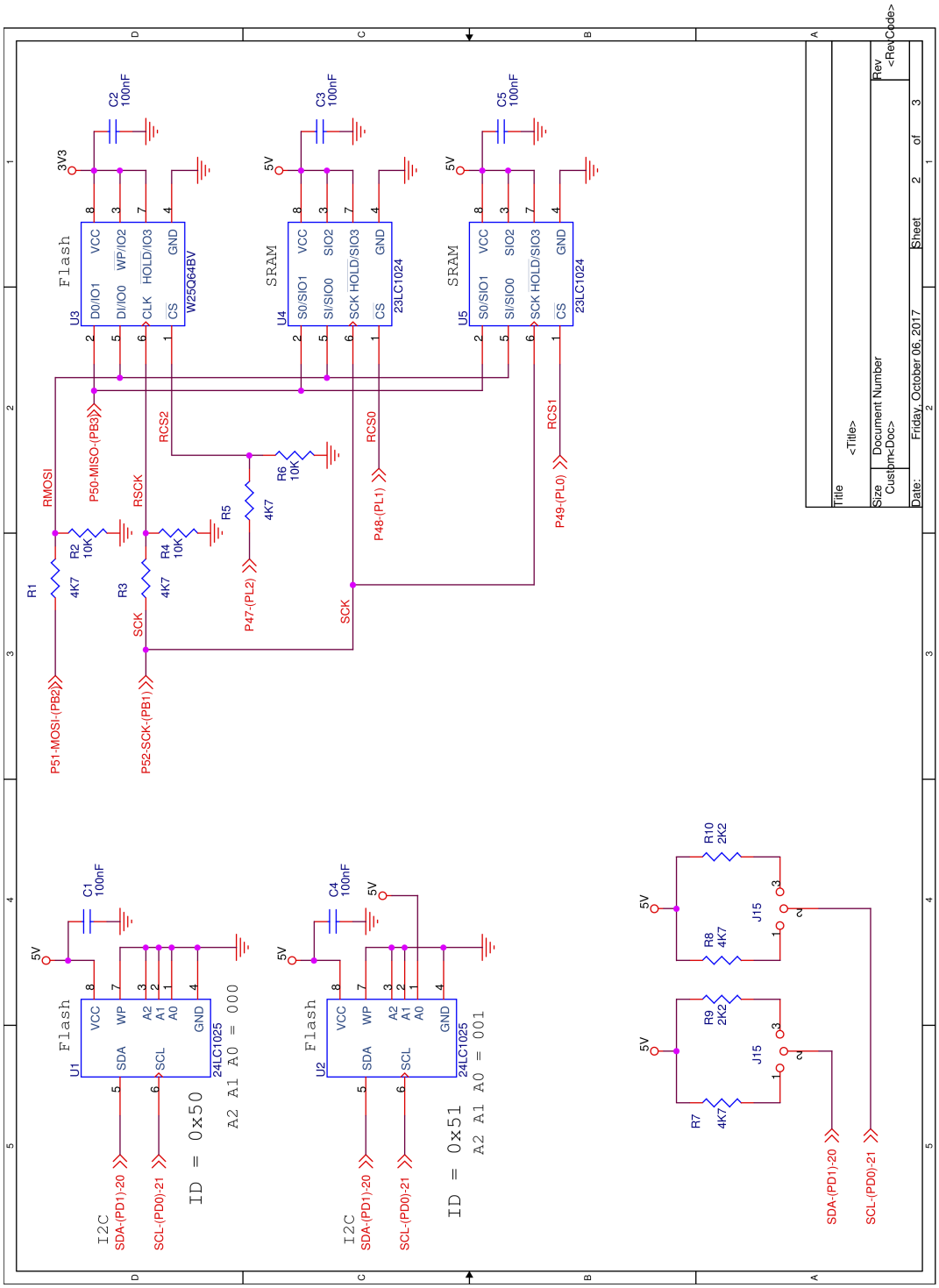
Apêndices

APÊNDICE A – Esquemático do Protótipo

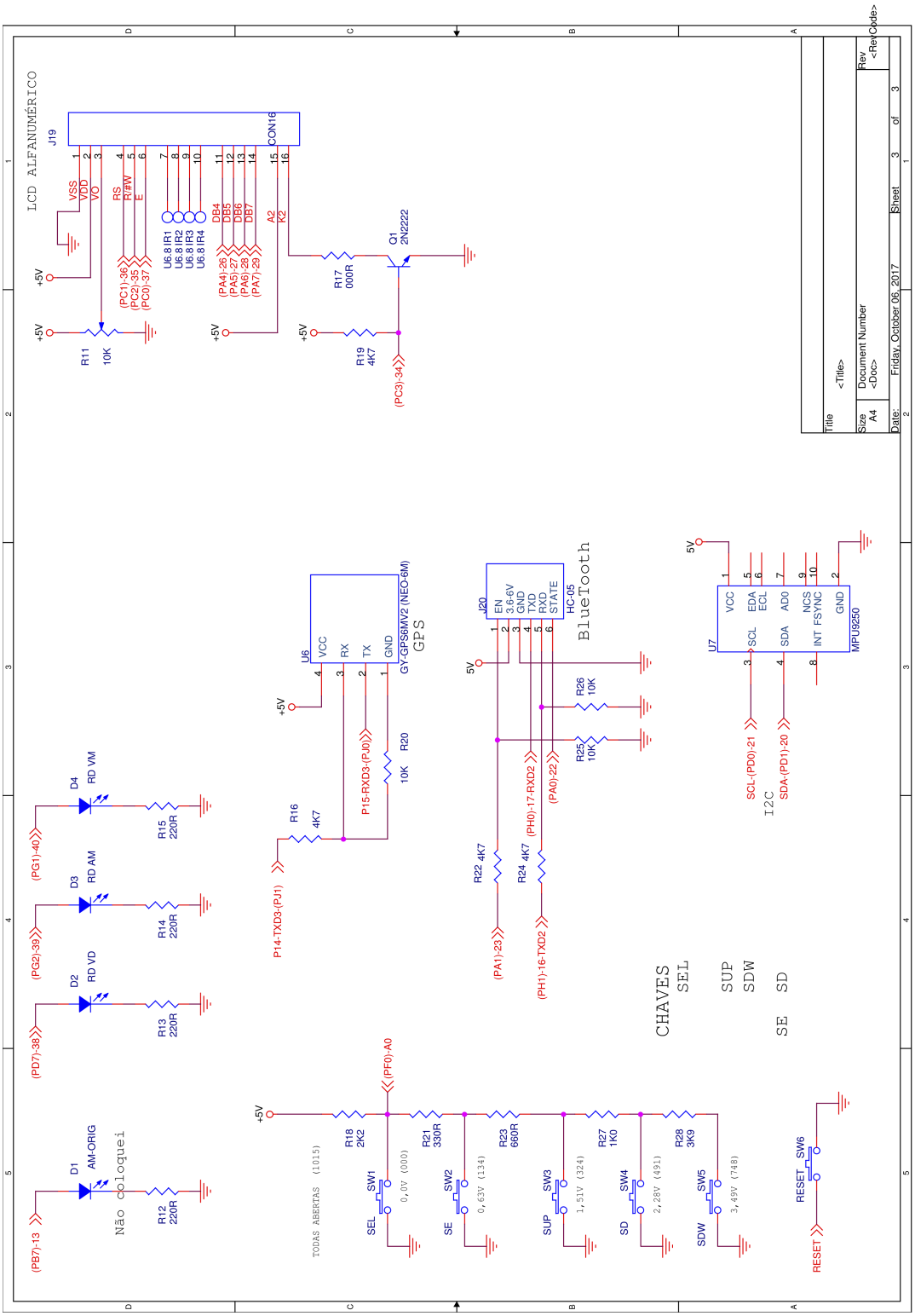
Conexão com Arduino MEGA



Memórias


Flash
W25Q64BV
SRAM
23LC1024
24LC1025
I2C
ID = 0x50
A2 A1 A0 = 000
ID = 0x51
A2 A1 A0 = 001
RMOSI
RSCK
SCK
RCS0
RCS1
RCS2
P50-MISO-(PB3)
P51-MOSI-(PB2)
P52-SCK-(PB1)
P47-(PL2)
P48-(PL1)
P49-(PL0)
SDA-(PD1)-20
SCL-(PD0)-21
Title
<Title>
Size
Custom
Document Number
<Doc>
Rev
<RevCode>
Date:
Friday, October 06, 2017
Sheet
2
of
3

Display, sensores e periféricos.



APÊNDICE B – Códigos

Código principal da plataforma Caixa-preta.

```
1
2 #include <I2C16.h>  // biblioteca para uso da biblioteca da EEPROM
3 #include <EEPROM_24XX1025.h> // biblioteca para o uso da memoria
      EEPROM
4
5 #include <LiquidCrystal.h> // biblioteca para o uso do display LCD
6 #include "MPU9250_t.h"  // driver do MARG/IMU MPU 9250
7 #include <SPI.h>
8 #include <Wire.h>
9
10 MPU9250 mpu;
11 EEPROM_24XX1025 eeprom (0, 0);
12
13 // 1: liga mensagens de debug; 0: desliga
14 #define DebugOn 0
15
16 #define TRUE  1
17 #define FALSE 0
18
19 // mensagem a ser recebida pelo GPS
20 typedef struct {
21   uint8_t   cls;
22   uint8_t   id;
23   uint16_t  len;
24   uint32_t  iTOW;
25   int32_t   fTOW;
26   int16_t   week;
27   uint8_t   gpsFix;
28   uint8_t   flags;
29   int32_t   ecefX;
30   int32_t   ecefY;
31   int32_t   ecefZ;
32   uint32_t  pAcc;
33   int32_t   ecefVX;
34   int32_t   ecefVY;
35   int32_t   ecefVZ;
36   uint32_t  sAcc;
37   uint16_t  pDOP;
38   uint8_t   reserved1;
39   uint8_t   numSV;
```

```
40   uint32_t  reserved2;
41 } navSol_t;
42
43 // Mensagem de requisicao de dados GPS
44 const uint8_t Poll_NAV_SOL[] = {0xB5, 0x62,
45                                 0x01, 0x06,
46                                 0x00, 0x00,
47                                 0x07, 0x16};
48
49
50 ////////////////// CHAVES //////////////////////////
51 // Valores lidos pelo conversor AD0
52 #define  SW_VAL_DIR    491
53 #define  SW_VAL_CIMA   324
54 #define  SW_VAL_BAIXO  748
55 #define  SW_VAL_ESQ    134
56 #define  SW_VAL_SEL    0
57 #define  SW_VAL_NADA   1015
58
59 // Parametros para leitura das chaves
60 #define  SW_TOL     20    // tolerancia para identificar chave
61 #define  SW_REP     20    // repeticoes de leitura - evitar ruido
62
63 // Codigos para as chaves
64 #define  SW_NADA   0
65 #define  SW_CIMA   1
66 #define  SW_ESQ    2
67 #define  SW_SEL    3
68 #define  SW_DIR    4
69 #define  SW_BAIXO  5
70 #define  SW_NAOSEI 100
71
72 uint8_t velha = 0; //SW_NADA;  // Guardar valor antigo da chave
73
74 ////////////////// LCD //////////////////////////////////////////
75 //                RS,  E, D4, D5, D6, D7 --> conexoes do LCD
76 LiquidCrystal lcd(36, 37, 26, 27, 28, 29);
77 const uint8_t LCD_RW = 35;
78
79 ////////////////// ADC0 /////////////////////////////////////////
80 const uint8_t ADC0 = A0;    // select the input pin for the selection
       button
81
82 /////////////// LEDS ////////////////////////////////////////////
83 const uint8_t lcd_bl = 34;  //Led amarelo original
84 const uint8_t led_vm = 38;
85 const uint8_t led_am = 39;
```

```
86 const uint8_t led_vd = 40;
87
88 ////////////////////////// SRAM ////////////////////////////
89 const uint8_t CSpin = 48;
90
91 // parametros de parada da rotina principal
92 // 2g para ter uma batida
93 const uint16_t dax = 16384, day = 16384, daz = 16384;
94 // 500 graus por segundo para ter uma derrapagem
95 const uint16_t dgx = 8192, dgy = 8192, dgz = 8192;
96
97 void setup() {
98
99   pinMode(LCD_RW, OUTPUT);
100   digitalWrite(LCD_RW, LOW);    // RW=0 <<<==== Importante
101   pinMode(CSpin, OUTPUT);
102
103   // determina o numero de colunas e linhas no display
104   lcd.begin(16, 2);
105
106   // escreve no display LCD
107   lcd.print("hello, world!");
108
109   Serial.begin(115200);
110   Serial3.begin(115200);
111   Wire.begin();
112   SPI.begin();
113   pinMode(lcd_bl, OUTPUT);
114   pinMode(led_vm, OUTPUT);
115   pinMode(led_am, OUTPUT);
116   pinMode(led_vd, OUTPUT);
117
118   digitalWrite(lcd_bl, LOW);    // led_or=0
119   digitalWrite(led_vm, LOW);    // led_or=0
120   digitalWrite(led_am, LOW);    // led_or=0
121   digitalWrite(led_vd, LOW);    // led_or=0
122
123   digitalWrite(led_vm, HIGH);    // led_vm=1
124   delay(500);
125   digitalWrite(led_vm, LOW);    // led_or=0
126
127   digitalWrite(led_am, HIGH);    // led_or=0
128   delay(500);
129   digitalWrite(led_am, LOW);    // led_or=0
130
131   digitalWrite(led_vd, HIGH);    // led_or=0
132   delay(500);
```

```
133   digitalWrite(led_vd, LOW);    // led_or=0
134
135   digitalWrite(lcd_bl, HIGH);    // led_or=0
136
137   lcd.clear();
138
139   setup_mpu();
140 }
141
142 // Retorna codigo da chave acionada
143 int SW_qual(void){
144   int i, Valor_ADC0;
145
146  Valor_ADC0 = 0;
147  for (i = 0; i < SW_REP; i++){
148    Valor_ADC0 = Valor_ADC0 + analogRead(ADC0);
149  }
150  Valor_ADC0 = Valor_ADC0/SW_REP;
151
152  if (abs(Valor_ADC0 - SW_VAL_NADA)  < SW_TOL)  return(SW_NADA);
153  if (abs(Valor_ADC0 - SW_VAL_CIMA)  < SW_TOL)  return(SW_CIMA);
154  //if (abs(Valor_ADC0 - SW_VAL_ESQ)   < SW_TOL)  return(SW_ESQ);
155  if (abs(Valor_ADC0 - SW_VAL_SEL)   < SW_TOL)  return(SW_SEL);
156  //if (abs(Valor_ADC0 - SW_VAL_DIR)   < SW_TOL)  return(SW_DIR);
157  if (abs(Valor_ADC0 - SW_VAL_BAIXO) < SW_TOL)  return(SW_BAIXO);
158
159  return(SW_NAOSEI);
160
161 }
162
163 // Esperar uma nova chave
164 int SW_Wait_qual (void){
165   int  nova;
166   while (1){
167     nova = SW_qual();
168     if ( (nova != SW_NAOSEI) && (nova != velha) ){
169       if (velha == SW_NADA){  velha = nova;  return nova;  }
170       else                    velha = nova;
171     }
172   }
173 }
174
175 void loop() {
176
177   uint8_t chave; static uint8_t option = 0;
178   //lcd.clear();
179   lcd.setCursor(0,0);
```

```
180
181
182   switch (option){
183     case 0:    lcd.print(F("Testar IMU     1"));   break;
184     case 1:    lcd.print(F("Testar GPS     2"));   break;
185     case 2:    lcd.print(F("Rodar Rotina   3"));   break;
186     case 3:    lcd.print(F("Enviar Dados   4"));   break;
187     case 4:    lcd.print(F("Apagar Dados   5"));   break;
188     default:         lcd.clear();    break;
189   }
190
191   chave = SW_Wait_qual();
192
193   if (chave == SW_BAIXO){
194     option++;
195     if(option > 4) option &= 0x00;
196   }
197   else if (chave == SW_CIMA){
198     option--;
199     if(option > 4) option &= 0x04;
200   }
201   else if(chave == SW_SEL){
202     lcd.setCursor(1, 1);
203     switch (option){
204       case 0:    lcd.print(F("Data @ Serial  ")); testMPU();    break;
205       case 1:    lcd.print(F("Data @ Serial  ")); testGPS();    break;
206       case 2:    lcd.print(F("SRAM&EEPROM    ")); routine();    break;
207       case 3:    lcd.print(F("data2serial    "));   break;
208       case 4:    lcd.print(F("trashes data   "));   break;
209       default:         lcd.clear();    break;
210   }
211   }
212   if (DebugOn)   Serial.println(option);
213
214 }
215
216 void routine(){
217   int16_t vetor[288];    // 576 bytes de dados (32 amostras de 18 bytes
        )
218   int16_t mover[64];     // 128 bytes de dados para serem transferidos
219   uint8_t evento = 0, fim = 0;
220   uint16_t writes = 0, jj;
221   static uint8_t j = 0;         // contagem de amostras
222   static uint32_t eEnd = 0, start_add = 0;
223   uint32_t t_f, sEnd, evento_End;
224   loading();
225   lcd.clear();
```

```
226   lcd.print(F("Rodar Rotina   3"));
227
228   while(!fim){
229
230     do{
231       delayMicroseconds(10);
232     } while((micros() - t_f) < 10000);
233     // le dados do sensor MPU 9250
234     mpu.readMotionSensor( (vetor + (j*9 + 0)), (vetor + (j*9 + 1)), (
      vetor + (j*9 + 2)),
235                           (vetor + (j*9 + 3)), (vetor + (j*9 + 4)), (
      vetor + (j*9 + 5)),
236                           (vetor + (j*9 + 6)), (vetor + (j*9 + 7)), (
      vetor + (j*9 + 8)) );
237     t_f = micros();
238     j++;
239
240     if ( ((vetor + (j*9 + 0)) > dax) || ((vetor + (j*9 + 1)) > day) ||
      ((vetor + (j*9 + 2)) > daz) ){
241       evento = 1;
242     }
243
244     if ( ((vetor + (j*9 + 3)) > dgx) || ((vetor + (j*9 + 4)) > dgy) ||
      ((vetor + (j*9 + 5)) > dgz) ){
245       evento = 1;
246     }
247
248     if(j == 31){
249       write2sram(eEnd, vetor); // SRAM armazenara 7264 amostras
250
251       if (evento) {
252         fim = 1;
253         evento_End = eEnd;
254       }
255
256       eEnd += sizeof(vetor);
257       if(eEnd >= 0x1FFFF) eEnd -= 0x1FFFF;
258       lcd.clear();
259       lcd.print(eEnd);
260       lcd.setCursor(0, 1);
261       lcd.print("escritas ");
262       lcd.setCursor(12, 1);
263       lcd.print(writes + 1);
264       j = 0; writes++;
265     }
266   } // fim do while loop
267
```

```
268   // loop para preencher o restante da memoria no pos-evento
269   for(uint8_t ii = 0; ii < 114; ii++){
270
271     do{
272       delayMicroseconds(10);
273     } while((micros() - t_f) < 10000);
274     // le dados do sensor MPU 9250
275     mpu.readMotionSensor( (vetor + (j*9 + 0)), (vetor + (j*9 + 1)), (
      vetor + (j*9 + 2)),
276                           (vetor + (j*9 + 3)), (vetor + (j*9 + 4)), (
      vetor + (j*9 + 5)),
277                           (vetor + (j*9 + 6)), (vetor + (j*9 + 7)), (
      vetor + (j*9 + 8)) );
278     t_f = micros();
279     j++;
280
281     if(j == 31){
282       write2sram(eEnd, vetor); // SRAM armazenara 7264 amostras
283       eEnd += sizeof(vetor);
284       if(eEnd >= 0x1FFFF) eEnd -= 0x1FFFF;
285       lcd.clear();
286       lcd.print(eEnd);
287       lcd.setCursor(0, 1);
288       lcd.print("escritas ");
289       lcd.setCursor(12, 1);
290       lcd.print(writes + 1);
291       j = 0; writes++;
292     }
293
294   } // fim do for loop
295
296   // definir endereco do inicio dos dados
297   start_add = evento_End - 32*18*114;      // 32*18*114 representa os
       dados do pre-evento
298   if(start_add < 0) start_add += 0x1FFFF; // garante que o inicio tem
       um valor valido
299
300   // sao 1021 pacotes de 128 bytes + 64 bytes extras
301   for(jj = 0; j < 1022; jj++){
302     readSram(start_add + jj*128, mover, sizeof(mover));  // le da SRAM
       128 bytes sequencialmente
303     eeprom.write(jj*128, mover, sizeof(mover)); // escreve na EEPROM
      128 sequencialmente (max size)
304     if(start_add > 0x1FFFF) start_add -= 0x1FFFF;
305   }
306   // transfere os 64 bytes restantes
307   readSram(start_add + jj*128, mover, 64);
```

```
308   eeprom.write(jj*128, mover, 64);
309
310   return;
311 }
312
313 void apagarEEPROM(){
314
315   for(uint32_t j = 0; j < 0x20000; j++){
316     eeprom.write(j, NULL, 1);
317   }
318 }
319
320 void transEEPROM(){
321
322   uint16_t carga[9];
323   for(uint32_t j = 0; j < 0x20000; j += 18){
324
325     eeprom.read(j, carga, sizeof(carga));
326
327     Serial.print("Raw:");
328     for(uint8_t i = 0; i < 5; i+=2){
329       Serial.print(((int16_t)carga[i] << 8) | carga[i+1]);
330     }
331
332   }
333
334 }
335
336 void testGPS(){
337   navSol_t gps;
338   uint8_t navsol[60], CK[2] = {0x00, 0x00}, j;
339   float veloc, posi;
340   loading();
341   lcd.clear();
342   lcd.print(F("Testar GPS     2"));
343   while(1){
344     // para encerrar o loop pressione qq botao
345     if(analogRead(ADC0) < 1000){
346       lcd.clear();
347       return;
348     }
349
350     Serial3.write(Poll_NAV_SOL, sizeof(Poll_NAV_SOL));
351     delay(1000);
352
353     uint8_t i = 0;
354     while(Serial3.available()){
```

```
355         navsol[i] = Serial3.read();
356         i++;
357       }
358
359       for(j = 2; j < (sizeof(navSol_t) + 2); j ++){
360         ((uint8_t *)(&gps))[j - 2] = navsol[j];
361
362         CK[0] += navsol[j];
363         CK[1] += CK[0];
364       }
365
366       if (CK[0] == navsol[j++] && CK[1] == navsol[j++]){
367         if(DebugOn) Serial.println(F("DEU BOM"));
368       }
369       memset(CK, 0, 2);
370
371       veloc = sqrt(pow((float)(gps.ecefVX), 2) + pow((float)(gps.ecefVY)
      , 2) + pow((float)(gps.ecefVZ), 2));
372       posi = sqrt(pow((float)(gps.ecefX), 2) + pow((float)(gps.ecefY), 2
      ) + pow((float)(gps.ecefZ), 2));
373       Serial.print(F("Centro do Mundo: ")); Serial.print(posi/100000, 0)
      ; Serial.print(" km");
374       Serial.print(" +"); Serial.print("- "); Serial.print((float)(gps.
      pAcc/100000.0), 0);  Serial.println(" km");
375       Serial.print(F("|V|: ")); Serial.print(veloc/100000, 2); Serial.
      println(" km/h");
376       Serial.print(F("iTOW: ")); Serial.print(gps.iTOW); Serial.print("\
      tweek: "); Serial.println(gps.week);
377       lcd.setCursor(0, 1);
378
379       // ver datasheet
380       switch (gps.gpsFix){
381         case (0x00):
382           lcd.print("No GPS fix       ");
383           break;
384         case (0x01):
385           lcd.print(F("DeadReck only    "));
386           break;
387         case (0x02):
388           lcd.print(F("2D-Fix           "));
389           break;
390         case (0x03):
391           lcd.print(F("3D-Fix           "));
392           break;
393         case (0x04):
394           lcd.print(F("GPS+DeadReck     "));
395           break;
```

```
396         case (0x05):
397           lcd.print(F("Time fix only    "));
398           break;
399         default:
400           lcd.print("DEU BOM NAO      ");
401           break;
402       }
403       delay(900);
404     }
405 }
406
407 void testMPU(){
408   int16_t dados[9];
409   uint32_t start;
410   loading();
411   lcd.clear();
412   lcd.print(F("Testar IMU      1"));
413   while(1){
414     // para encerrar o loop pressione qq botao
415     if(analogRead(ADC0) < 1000){
416       lcd.clear();
417       return;
418     }
419     start = micros();
420     mpu.readMotionSensor((dados + 0), (dados + 1), (dados + 2),
421                          (dados + 3), (dados + 4), (dados + 5),
422                          (dados + 6), (dados + 7), (dados + 8));
423
424     Serial.print(F("Raw:"));
425     for(uint8_t i = 0; i < 9; i++){
426       if(i > 5) Serial.print((int16_t)((float)dados[i]*mpu.
      factoryMagCalibration[i - 6]));
427       else      Serial.print(dados[i]);
428       if(i != 8) Serial.print(',');
429     }
430     Serial.println();
431     do{
432       delayMicroseconds(10);
433     } while((micros() - start) < 10000);
434   }
435 }
436
437 void setup_mpu(){
438   uint8_t c;
439
440   c = mpu.readByte(MPU9250_ADDRESS, WHO_AM_I_MPU9250);
441
```

```
442   if (c == 0x73){
443     if (DebugOn)  Serial.println(F("MPU9250 is online..."));
444     mpu.calibrateMPU9250(mpu.gyroBias, mpu.accelBias);
445     mpu.initMPU9250();
446
447     c = mpu.readByte(AK8963_ADDRESS, WHO_AM_I_AK8963);
448
449     if(c != 0x48){
450       if (DebugOn)  Serial.println(F("Communication failed, abort!"));
451       Serial.flush();
452       abort();
453     }
454
455     mpu.initAK8963(mpu.factoryMagCalibration);
456
457     if (DebugOn)
458     {
459       Serial.println(F("Calibration values: "));
460       Serial.print(F("X-Axis factory sensitivity adjustment value "));
461       Serial.println(mpu.factoryMagCalibration[0], 2);
462       Serial.print(F("Y-Axis factory sensitivity adjustment value "));
463       Serial.println(mpu.factoryMagCalibration[1], 2);
464       Serial.print(F("Z-Axis factory sensitivity adjustment value "));
465       Serial.println(mpu.factoryMagCalibration[2], 2);
466     }
467   }
468   else {
469     if (DebugOn)  Serial.print(F("Something is wrong..."));
470     return;
471   }
472 }
473
474 void loading(){
475   randomSeed(analogRead(A1));
476   lcd.clear();
477   lcd.print(F("Loading..."));
478   for(uint8_t i = 0; i < 17; i++){
479     lcd.setCursor(i, 1);
480     lcd.print('X');
481     delay((i*random(50, 250))/random(4, 8));
482     digitalWrite(led_am, !digitalRead(led_am));    // led_or=0
483   }
484   digitalWrite(led_am, LOW);    // led_or=0
485 }
```

Listing B.1 – Programa principal

Código de escrita e leitura na memória SRAM.

```
1 // modos de operacao do SRAM
2 #define RDMR 0x05 // read Mode Register
3 #define WRMR 0x01 // write to Mode Register
4 #define BYMD 0x00 // Byte Mode
5 #define PGMD 0x80 // Page Mode
6 #define SQMD 0x40 // Sequential Mode
7 #define READ 0x03 // Read SRAM
8 #define WRITE 0x02 // Write to SRAM
9
10 // the best mode to write and read from SRAM is the sequencial mode
11 // which is default.
12
13 uint8_t read_MODE(){
14   uint8_t mode = 2;
15   digitalWrite(CSpin, LOW);
16   delay(5);
17   SPI.transfer(RDMR);
18   mode = SPI.transfer(0x00);
19   digitalWrite(CSpin, HIGH);
20   return mode;
21 }
22
23 void set_MODE(uint8_t mode){
24   digitalWrite(CSpin, LOW);
25   delay(10);
26   SPI.transfer(WRMR);
27   SPI.transfer(mode);
28   digitalWrite(CSpin, HIGH);
29 }
30
31 void write2sram(uint32_t addr, int16_t *buff){
32
33   //uint16_t i;
34   digitalWrite(CSpin, LOW);
35   SPI.transfer(WRITE);
36   SPI.transfer((uint8_t)(addr >> 16));
37   SPI.transfer((uint8_t)(addr >> 8));
38   SPI.transfer((uint8_t)addr);
39   for (uint16_t i = 0; i < 288; i++){
40     SPI.transfer((uint8_t)(buff[i] >> 8));  // sends MSByte as
     unsigned 8bit
41     SPI.transfer((uint8_t)buff[i]);         // sends LSByte as
     unsigned 8bit
42   }
43   digitalWrite(CSpin, HIGH);
```

```
44 }
45 // function to read from the SRAM (signed 16bits-byte)
46 // pay attantion to read from right address as each write takes 2
       bytes
47 void readSram(uint32_t addr, int16_t *buff, uint32_t numBytes){
48   digitalWrite(CSpin, LOW);
49   SPI.transfer(READ);
50   SPI.transfer((uint8_t)(addr >> 16));
51   SPI.transfer((uint8_t)(addr >> 8));
52   SPI.transfer((uint8_t)addr);
53   for (uint16_t i = 0; i < numBytes; i++) {
54     buff[i] = (int16_t)(SPI.transfer(0x00) << 8) | SPI.transfer(0x00);
55     // reads two bytes sequentially to form a signed 16bit number
56   }
57   digitalWrite(CSpin, HIGH);
58 }
```

Listing B.2 – Códigos de uso da memória SRAM

Anexos

ANEXO A – *Driver* do sensor MPU-9250

Código do *driver* do MPU-9250 disponibilizado pela SparkFun.

```
/*
 Note: The MPU9250 is an I2C sensor and uses the Arduino Wire library.
 Because the sensor is not 5V tolerant, we are using a 3.3 V 8 MHz Pro Mini or
 a 3.3 V Teensy 3.1. We have disabled the internal pull-ups used by the Wire
 library in the Wire.h/twi.c utility file. We are also using the 400 kHz fast
 I2C mode by setting the TWI_FREQ to 400000L /twi.h utility file.
 */
#ifndef _MPU9250_H_
#define _MPU9250_H_

#include <SPI.h>
#include <Wire.h>
#include "util/crc16.h"
#include <EEPROM.h>

#define SERIAL_DEBUG true

// See also MPU-9250 Register Map and Descriptions, Revision 4.0,
// RM-MPU-9250A-00, Rev. 1.4, 9/9/2013 for registers not listed in above
// document; the MPU9250 and MPU9150 are virtually identical but the latter has
// a different register map

//Magnetometer Registers
#define AK8963_ADDRESS   0x0C
#define WHO_AM_I_AK8963  0x00 // (AKA WIA) should return 0x48
#define INFO             0x01
#define AK8963_ST1       0x02  // data ready status bit 0
#define AK8963_XOUT_L    0x03  // data
#define AK8963_XOUT_H    0x04
#define AK8963_YOUT_L    0x05
#define AK8963_YOUT_H    0x06
#define AK8963_ZOUT_L    0x07
#define AK8963_ZOUT_H    0x08
#define AK8963_ST2       0x09  // Data overflow bit 3 and data read error status bit 2
#define AK8963_CNTL      0x0A  // Power down (0000), single-
```

```
      measurement (0001), self-test (1000) and Fuse ROM (1111) modes on
      bits 3:0
36 #define AK8963_ASTC      0x0C  // Self test control
37 #define AK8963_I2CDIS    0x0F  // I2C disable
38 #define AK8963_ASAX      0x10  // Fuse ROM x-axis sensitivity
      adjustment value
39 #define AK8963_ASAY      0x11  // Fuse ROM y-axis sensitivity
      adjustment value
40 #define AK8963_ASAZ      0x12  // Fuse ROM z-axis sensitivity
      adjustment value
41
42 #define SELF_TEST_X_GYRO 0x00
43 #define SELF_TEST_Y_GYRO 0x01
44 #define SELF_TEST_Z_GYRO 0x02
45
46 /*#define X_FINE_GAIN      0x03 // [7:0] fine gain
47 #define Y_FINE_GAIN      0x04
48 #define Z_FINE_GAIN      0x05
49 #define XA_OFFSET_H      0x06 // User-defined trim values for
      accelerometer
50 #define XA_OFFSET_L_TC   0x07
51 #define YA_OFFSET_H      0x08
52 #define YA_OFFSET_L_TC   0x09
53 #define ZA_OFFSET_H      0x0A
54 #define ZA_OFFSET_L_TC   0x0B */
55
56 #define SELF_TEST_X_ACCEL 0x0D
57 #define SELF_TEST_Y_ACCEL 0x0E
58 #define SELF_TEST_Z_ACCEL 0x0F
59
60 #define SELF_TEST_A       0x10
61
62 #define XG_OFFSET_H       0x13  // User-defined trim values for
      gyroscope
63 #define XG_OFFSET_L       0x14
64 #define YG_OFFSET_H       0x15
65 #define YG_OFFSET_L       0x16
66 #define ZG_OFFSET_H       0x17
67 #define ZG_OFFSET_L       0x18
68 #define SMPLRT_DIV        0x19
69 #define CONFIG            0x1A
70 #define GYRO_CONFIG       0x1B
71 #define ACCEL_CONFIG      0x1C
72 #define ACCEL_CONFIG2     0x1D
73 #define LP_ACCEL_ODR      0x1E
74 #define WOM_THR           0x1F
75
```

```
76 // Duration counter threshold for motion interrupt generation, 1 kHz
       rate,
77 // LSB = 1 ms
78 #define MOT_DUR             0x20
79 // Zero-motion detection threshold bits [7:0]
80 #define ZMOT_THR            0x21
81 // Duration counter threshold for zero motion interrupt generation, 16
       Hz rate,
82 // LSB = 64 ms
83 #define ZRMOT_DUR           0x22
84
85 #define FIFO_EN             0x23
86 #define I2C_MST_CTRL        0x24
87 #define I2C_SLV0_ADDR       0x25
88 #define I2C_SLV0_REG        0x26
89 #define I2C_SLV0_CTRL       0x27
90 #define I2C_SLV1_ADDR       0x28
91 #define I2C_SLV1_REG        0x29
92 #define I2C_SLV1_CTRL       0x2A
93 #define I2C_SLV2_ADDR       0x2B
94 #define I2C_SLV2_REG        0x2C
95 #define I2C_SLV2_CTRL       0x2D
96 #define I2C_SLV3_ADDR       0x2E
97 #define I2C_SLV3_REG        0x2F
98 #define I2C_SLV3_CTRL       0x30
99 #define I2C_SLV4_ADDR       0x31
100 #define I2C_SLV4_REG        0x32
101 #define I2C_SLV4_DO         0x33
102 #define I2C_SLV4_CTRL       0x34
103 #define I2C_SLV4_DI         0x35
104 #define I2C_MST_STATUS      0x36
105 #define INT_PIN_CFG         0x37
106 #define INT_ENABLE          0x38
107 #define DMP_INT_STATUS      0x39  // Check DMP interrupt
108 #define INT_STATUS          0x3A
109 #define ACCEL_XOUT_H        0x3B
110 #define ACCEL_XOUT_L        0x3C
111 #define ACCEL_YOUT_H        0x3D
112 #define ACCEL_YOUT_L        0x3E
113 #define ACCEL_ZOUT_H        0x3F
114 #define ACCEL_ZOUT_L        0x40
115 #define TEMP_OUT_H          0x41
116 #define TEMP_OUT_L          0x42
117 #define GYRO_XOUT_H         0x43
118 #define GYRO_XOUT_L         0x44
119 #define GYRO_YOUT_H         0x45
120 #define GYRO_YOUT_L         0x46
```

```
121 #define GYRO_ZOUT_H         0x47
122 #define GYRO_ZOUT_L         0x48
123 #define EXT_SENS_DATA_00    0x49
124 #define EXT_SENS_DATA_01    0x4A
125 #define EXT_SENS_DATA_02    0x4B
126 #define EXT_SENS_DATA_03    0x4C
127 #define EXT_SENS_DATA_04    0x4D
128 #define EXT_SENS_DATA_05    0x4E
129 #define EXT_SENS_DATA_06    0x4F
130 #define EXT_SENS_DATA_07    0x50
131 #define EXT_SENS_DATA_08    0x51
132 #define EXT_SENS_DATA_09    0x52
133 #define EXT_SENS_DATA_10    0x53
134 #define EXT_SENS_DATA_11    0x54
135 #define EXT_SENS_DATA_12    0x55
136 #define EXT_SENS_DATA_13    0x56
137 #define EXT_SENS_DATA_14    0x57
138 #define EXT_SENS_DATA_15    0x58
139 #define EXT_SENS_DATA_16    0x59
140 #define EXT_SENS_DATA_17    0x5A
141 #define EXT_SENS_DATA_18    0x5B
142 #define EXT_SENS_DATA_19    0x5C
143 #define EXT_SENS_DATA_20    0x5D
144 #define EXT_SENS_DATA_21    0x5E
145 #define EXT_SENS_DATA_22    0x5F
146 #define EXT_SENS_DATA_23    0x60
147 #define MOT_DETECT_STATUS   0x61
148 #define I2C_SLV0_DO         0x63
149 #define I2C_SLV1_DO         0x64
150 #define I2C_SLV2_DO         0x65
151 #define I2C_SLV3_DO         0x66
152 #define I2C_MST_DELAY_CTRL  0x67
153 #define SIGNAL_PATH_RESET   0x68
154 #define MOT_DETECT_CTRL     0x69
155 #define USER_CTRL           0x6A  // Bit 7 enable DMP, bit 3 reset DMP
156 #define PWR_MGMT_1          0x6B // Device defaults to the SLEEP mode
157 #define PWR_MGMT_2          0x6C
158 #define DMP_BANK            0x6D  // Activates a specific bank in the
      DMP
159 #define DMP_RW_PNT          0x6E  // Set read/write pointer to a
      specific start address in specified DMP bank
160 #define DMP_REG             0x6F  // Register in DMP from which to read
       or to which to write
161 #define DMP_REG_1           0x70
162 #define DMP_REG_2           0x71
163 #define FIFO_COUNTH         0x72
164 #define FIFO_COUNTL         0x73
```

```
165 #define FIFO_R_W             0x74
166 #define WHO_AM_I_MPU9250     0x75 // Should return 0x71
167 #define XA_OFFSET_H          0x77
168 #define XA_OFFSET_L          0x78
169 #define YA_OFFSET_H          0x7A
170 #define YA_OFFSET_L          0x7B
171 #define ZA_OFFSET_H          0x7D
172 #define ZA_OFFSET_L          0x7E
173
174 // Using the MPU-9250 breakout board, ADO is set to 0
175 // Seven-bit device address is 110100 for ADO = 0 and 110101 for ADO =
        1
176 #define ADO 0
177 #if ADO
178 #define MPU9250_ADDRESS 0x69  // Device address when ADO = 1
179 #else
180 #define MPU9250_ADDRESS 0x68  // Device address when ADO = 0
181 #define AK8963_ADDRESS  0x0C   // Address of magnetometer
182 #endif // AD0
183
184 #define READ_FLAG 0x80
185 #define NOT_SPI -1
186 #define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the MPU-9250
187 //#define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the MPU
        -9250
188 #define SPI_MODE SPI_MODE3
189
190  #define MPU9250_CAL_SIZE 68
191  #define MPU9250_CAL_EEADDR 60
192
193 class MPU9250
194 {
195   protected:
196     // Set initial input parameters
197     enum Ascale
198     {
199       AFS_2G = 0,
200       AFS_4G,
201       AFS_8G,
202       AFS_16G
203     };
204
205     enum Gscale {
206       GFS_250DPS = 0,
207       GFS_500DPS,
208       GFS_1000DPS,
209       GFS_2000DPS
```

```
210     };
211
212     enum Mscale {
213       MFS_14BITS = 0, // 0.60 mG per LSB
214       MFS_16BITS      // 0.15 mG per LSB
215
216     };
217
218     enum M_MODE {
219       M_8HZ = 0x02,  // 8 Hz update
220       M_100HZ = 0x06 // 100 Hz continuous magnetometer
221     };
222
223     // TODO: Add setter methods for this hard coded stuff
224     // Specify sensor full scale
225     uint8_t Gscale = GFS_2000DPS;
226     uint8_t Ascale = AFS_4G;
227     // Choose either 14-bit or 16-bit magnetometer resolution
228     uint8_t Mscale = MFS_16BITS;
229
230     // 2 for 8 Hz, 6 for 100 Hz continuous magnetometer data read
231     uint8_t Mmode = M_100HZ;
232
233     // SPI chip select pin
234     int8_t _csPin;
235
236     uint8_t writeByteWire(uint8_t, uint8_t, uint8_t);
237     uint8_t writeByteSPI(uint8_t, uint8_t);
238     uint8_t readByteSPI(uint8_t subAddress);
239     uint8_t readByteWire(uint8_t address, uint8_t subAddress);
240     bool magInit();
241     void kickHardware();
242     void select();
243     void deselect();
244 // TODO: Remove this next line
245 public:
246     uint8_t ak8963WhoAmI_SPI();
247
248   public:
249     float pitch, yaw, roll;
250     float temperature;   // Stores the real internal chip temperature
      in Celsius
251     int16_t tempCount;   // Temperature raw count output
252     uint32_t delt_t = 0; // Used to control display output rate
253
254     uint32_t count = 0, sumCount = 0; // used to control display
      output rate
```

```
255     float deltat = 0.0f, sum = 0.0f;  // integration interval for both
        filter schemes
256     uint32_t lastUpdate = 0, firstUpdate = 0; // used to calculate
       integration interval
257     uint32_t Now = 0;         // used to calculate integration interval
258
259     int16_t gyroCount[3];   // Stores the 16-bit signed gyro sensor
       output
260     int16_t magCount[3];    // Stores the 16-bit signed magnetometer
       sensor output
261     // Scale resolutions per LSB for the sensors
262     float aRes, gRes, mRes;
263     // Variables to hold latest sensor data values
264     //float ax, ay, az, gx, gy, gz, mx, my, mz, temp;
265     int16_t accel_temp_raw[4];
266     int16_t mag_raw[3];
267     int16_t gyro_raw[3];
268     uint8_t newdata;
269     float cal[16];
270
271     // Factory mag calibration and mag bias
272     float factoryMagCalibration[3] = {0, 0, 0}, factoryMagBias[3] = {0
       , 0, 0};
273     // Bias corrections for gyro, accelerometer, and magnetometer
274     float gyroBias[3]  = {0, 0, 0},
275           accelBias[3] = {0, 0, 0},
276           magBias[3]   = {0, 0, 0},
277           magScale[3]  = {0, 0, 0};
278     float selfTest[6];
279     // Stores the 16-bit signed accelerometer sensor output
280     int16_t accelCount[3];
281
282     // Public method declarations
283     MPU9250(int8_t csPin=NOT_SPI);
284     void getMres();
285     void getGres();
286     void getAres();
287     int readAccelData(int16_t *);
288     void readMotionSensor(int16_t *ax, int16_t *ay, int16_t *az, int16
       _t *gx, int16_t *gy, int16_t *gz, int16_t *mx, int16_t *my,int16_t
       *mz);
289     void update();
290     bool writeCalibration(const void *);
291     void getCalibration(float *, float *, float *);
292     bool available();
293     int readGyroData(int16_t *);
294     int readMagData(int16_t *);
```

```
295     void readTempData(int16_t *);
296     void updateTime();
297     void initAK8963(float *);
298     void initMPU9250();
299     void calibrateMPU9250(float * gyroBias, float * accelBias);
300     void MPU9250SelfTest(float * destination);
301     void magCalMPU9250(float * dest1, float * dest2);
302     uint8_t writeByte(uint8_t, uint8_t, uint8_t);
303     uint8_t readByte(uint8_t, uint8_t);
304     uint8_t readBytes(uint8_t, uint8_t, uint8_t, uint8_t *);
305     // TODO: make SPI/Wire private
306     uint8_t readBytesSPI(uint8_t, uint8_t, uint8_t *);
307     uint8_t readBytesWire(uint8_t, uint8_t, uint8_t, uint8_t *);
308     bool isInI2cMode() { return _csPin == -1; }
309     bool begin();
310 };  // class MPU9250
311
312 #endif // _MPU9250_H_
```

Listing A.1 – Arquivo *Header* do *Driver*

```
1 #include "MPU9250_t.h"
2
3 //
    ================================================================================
4 //====== Set of useful function to access acceleration. gyroscope,
    magnetometer,
5 //====== and temperature data
6 //
    ================================================================================
7
8 MPU9250::MPU9250(int8_t cspin /*=NOT_SPI*/) // Uses I2C communication
    by default
9 {
10   // Use hardware SPI communication
11   // If used with sparkfun breakout board
12   // https://www.sparkfun.com/products/13762 , change the pre-soldered
      JP2 to
13   // enable SPI (solder middle and left instead of middle and right)
     pads are
14   // very small and re-soldering can be very tricky. I2C highly
     recommended.
15   if ((cspin > NOT_SPI) && (cspin < NUM_DIGITAL_PINS))
16   {
17     _csPin = cspin;
```

```
18      SPI.begin();
19      pinMode(_csPin, OUTPUT);
20      deselect();
21    }
22    else
23    {
24      _csPin = NOT_SPI;
25
26      Wire.begin();
27      Wire.setClock(400000);
28    }
29  }
30
31  void MPU9250::readMotionSensor(int16_t *ax, int16_t *ay, int16_t *az,
        int16_t *gx, int16_t *gy, int16_t *gz, int16_t *mx, int16_t *my,
        int16_t *mz) {
32    if (readByte(MPU9250_ADDRESS, INT_STATUS) & 0x01) update();
33    *ax = accel_temp_raw[0];
34    *ay = accel_temp_raw[1];
35    *az = accel_temp_raw[2];
36
37    *mx = mag_raw[0];
38    *my = mag_raw[1];
39    *mz = mag_raw[2];
40    *gx = gyro_raw[0];
41    *gy = gyro_raw[1];
42    *gz = gyro_raw[2];
43  }
44
45  void MPU9250::update()
46  {
47    //static elapsedMillis msec;
48    //int32_t alt;
49
50    if (readAccelData(accel_temp_raw)) { // accel + mag
51      //Serial.println("accel+mag");
52    }
53    if (readMagData(mag_raw)) { // alt
54      //Serial.println("alt");
55    }
56    if (readGyroData(gyro_raw)) {  // gyro
57      //Serial.println("gyro");
58    }
59  }
60
61  void MPU9250::getMres()
62  {
```

```
63   switch (Mscale)
64   {
65     // Possible magnetometer scales (and their register bit settings)
       are:
66     // 14 bit resolution (0) and 16 bit resolution (1)
67     case MFS_14BITS:
68       mRes = 10.0f * 4912.0f / 8190.0f; // Proper scale to return
       milliGauss
69       break;
70     case MFS_16BITS:
71       mRes = (10.0f * 4912.0f / 32760.0f) / 1.5; // Proper scale to
       return milliGauss
72       break;
73
74   }
75 }
76
77 void MPU9250::getGres()
78 {
79   switch (Gscale)
80   {
81     // Possible gyro scales (and their register bit settings) are:
82     // 250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS (11).
83     // Here's a bit of an algorith to calculate DPS/(ADC tick) based
       on that
84     // 2-bit value:
85     case GFS_250DPS:
86       gRes = 250.0f / 32768.0f;
87       break;
88     case GFS_500DPS:
89       gRes = 500.0f / 32768.0f;
90       break;
91     case GFS_1000DPS:
92       gRes = 1000.0f / 32768.0f;
93       break;
94     case GFS_2000DPS:
95       gRes = 2000.0f / 32768.0f;
96       break;
97   }
98 }
99
100 void MPU9250::getAres()
101 {
102   switch (Ascale)
103   {
104     // Possible accelerometer scales (and their register bit settings)
       are:
```

```
105     // 2 Gs (00), 4 Gs (01), 8 Gs (10), and 16 Gs  (11).
106     // Here's a bit of an algorith to calculate DPS/(ADC tick) based
      on that
107     // 2-bit value:
108     case AFS_2G:
109       aRes = 2.0f / 32768.0f;
110       break;
111     case AFS_4G:
112       aRes = 4.0f / 32768.0f;
113       break;
114     case AFS_8G:
115       aRes = 8.0f / 32768.0f;
116       break;
117     case AFS_16G:
118       aRes = 16.0f / 32768.0f;
119       break;
120   }
121 }
122
123
124 int MPU9250::readAccelData(int16_t *destination)
125 {
126   uint8_t rawData[6];  // x/y/z accel register data stored here
127   // Read the six raw data registers into data array
128   readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
129
130   // Turn the MSB and LSB into a signed 16-bit value
131   destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ; // accel
       x
132   destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ; // accel
       y
133   destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ; // accel
       z
134   //destination[3] = ((int16_t)rawData[6] << 8) | rawData[7] ; //
       temperature
135
136   return 1;
137 }
138
139
140 int MPU9250::readGyroData(int16_t *destination)
141 {
142   uint8_t rawData[6];  // x/y/z gyro register data stored here
143   // Read the six raw data registers sequentially into data array
144   readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
145
146   // Turn the MSB and LSB into a signed 16-bit value
```

```
147   destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ;
148   destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
149   destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
150
151   return 1;
152 }
153
154 int MPU9250::readMagData(int16_t *destination)
155 {
156   // x/y/z gyro register data, ST2 register stored here, must read ST2
        at end
157   // of data acquisition
158   uint8_t rawData[7];
159   // Wait for magnetometer data ready bit to be set
160   if (1/*readByte(AK8963_ADDRESS, AK8963_ST1) & 0x01*/)
161   {
162     // Read the six raw data and ST2 registers sequentially into data
       array
163     readBytes(AK8963_ADDRESS, AK8963_XOUT_L, 7, &rawData[0]);
164     uint8_t c = rawData[6]; // End data read by reading ST2 register
165     // Check if magnetic sensor overflow set, if not then report data
166     if (!(c & 0x08))
167     {
168       // Turn the MSB and LSB into a signed 16-bit value
169       destination[0] = ((int16_t)rawData[1] << 8) | rawData[0];
170       // Data stored as little Endian
171       destination[1] = ((int16_t)rawData[3] << 8) | rawData[2];
172       destination[2] = ((int16_t)rawData[5] << 8) | rawData[4];
173
174       return 1;
175     }
176   }
177
178   return 0;
179 }
180
181 void MPU9250::readTempData(int16_t * destination)
182 {
183   uint8_t rawData[2]; // x/y/z gyro register data stored here
184   // Read the two raw data registers sequentially into data array
185   readBytes(MPU9250_ADDRESS, TEMP_OUT_H, 2, &rawData[0]);
186   // Turn the MSB and LSB into a 16-bit value
187   destination = ((int16_t)rawData[0] << 8) | rawData[1];
188 }
189
190 // Calculate the time the last update took for use in the quaternion
       filters
```

```
191 // TODO: This doesn't really belong in this class.
192 void MPU9250::updateTime()
193 {
194   Now = micros();
195
196   // Set integration time by time elapsed since last filter update
197   deltat = ((Now - lastUpdate) / 1000000.0f);
198   lastUpdate = Now;
199
200   sum += deltat; // sum for averaging filter update rate
201   sumCount++;
202 }
203
204 void MPU9250::initAK8963(float * destination)
205 {
206   // First extract the factory calibration for each magnetometer axis
207   uint8_t rawData[3];  // x/y/z gyro calibration data stored here
208   // TODO: Test this!! Likely doesn't work
209   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
      magnetometer
210   delay(10);
211   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x0F); // Enter Fuse ROM
      access mode
212   delay(10);
213
214   // Read the x-, y-, and z-axis calibration values
215   readBytes(AK8963_ADDRESS, AK8963_ASAX, 3, &rawData[0]);
216
217   // Return x-axis sensitivity adjustment values, etc.
218   destination[0] =  (float)(rawData[0] - 128) / 256. + 1.;
219   destination[1] =  (float)(rawData[1] - 128) / 256. + 1.;
220   destination[2] =  (float)(rawData[2] - 128) / 256. + 1.;
221   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
      magnetometer
222   delay(10);
223
224   // Configure the magnetometer for continuous read and highest
      resolution.
225   // Set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution in
      CNTL
226   // register, and enable continuous mode data acquisition Mmode (bits
        [3:0]),
227   // 0010 for 8 Hz and 0110 for 100 Hz sample rates.
228
229   // Set magnetometer data resolution and sample ODR
230   writeByte(AK8963_ADDRESS, AK8963_CNTL, Mscale << 4 | Mmode);
231   delay(10);
```

```
232 }
233
234 void MPU9250::initMPU9250()
235 {
236   // wake up device
237   // Clear sleep mode bit (6), enable all sensors
238   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
239   delay(100); // Wait for all registers to reset
240
241   // Get stable time source
242   // Auto select clock source to be PLL gyroscope reference if ready
       else
243   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
244   delay(200);
245
246   // Configure Gyro and Thermometer
247   // Disable FSYNC and set thermometer and gyro bandwidth to 41 and 42
       Hz,
248   // respectively;
249   // minimum delay time for this setting is 5.9 ms, which means sensor
       fusion
250   // update rates cannot be higher than 1 / 0.0059 = 170 Hz
251   // DLPF_CFG = bits 2:0 = 011; this limits the sample rate to 1000 Hz
       for both
252   // With the MPU9250, it is possible to get gyro sample rates of 32
       kHz (!),
253   // 8 kHz, or 1 kHz
254   writeByte(MPU9250_ADDRESS, CONFIG, 0x03);
255
256   // Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
257   // Use a 200 Hz rate; a rate consistent with the filter update rate
258   // determined inset in CONFIG above.
259   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x04);
260
261   // Set gyroscope full scale range
262   // Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values are
263   // left-shifted into positions 4:3
264
265   // get current GYRO_CONFIG register value
266   uint8_t c = readByte(MPU9250_ADDRESS, GYRO_CONFIG);
267   // c = c & ~0xE0; // Clear self-test bits [7:5]
268   c = c & ~0x02; // Clear Fchoice bits [1:0]
269   c = c & ~0x18; // Clear AFS bits [4:3]
270   c = c | Gscale << 3; // Set full scale range for the gyro
271   // Set Fchoice for the gyro to 11 by writing its inverse to bits 1:0
       of
272   // GYRO_CONFIG
```

```
273   // c =| 0x00;
274   // Write new GYRO_CONFIG value to register
275   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c );
276
277   // Set accelerometer full-scale range configuration
278   // Get current ACCEL_CONFIG register value
279   c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG);
280   // c = c & ~0xE0; // Clear self-test bits [7:5]
281   c = c & ~0x18;  // Clear AFS bits [4:3]
282   c = c | Ascale << 3; // Set full scale range for the accelerometer
283   // Write new ACCEL_CONFIG register value
284   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c);
285
286   // Set accelerometer sample rate configuration
287   // It is possible to get a 4 kHz sample rate from the accelerometer
       by
288   // choosing 1 for accel_fchoice_b bit [3]; in this case the
       bandwidth is
289   // 1.13 kHz
290   // Get current ACCEL_CONFIG2 register value
291   c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG2);
292   c = c & ~0x0F; // Clear accel_fchoice_b (bit 3) and A_DLPFG (bits
       [2:0])
293   c = c | 0x03;  // Set accelerometer rate to 1 kHz and bandwidth to
       41 Hz
294   // Write new ACCEL_CONFIG2 register value
295   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c);
296   // The accelerometer, gyro, and thermometer are set to 1 kHz sample
       rates,
297   // but all these rates are further reduced by a factor of 5 to 200
       Hz because
298   // of the SMPLRT_DIV setting
299
300   // Configure Interrupts and Bypass Enable
301   // Set interrupt pin active high, push-pull, hold interrupt pin
       level HIGH
302   // until interrupt cleared, clear on read of INT_STATUS, and enable
303   // I2C_BYPASS_EN so additional chips can join the I2C bus and all
       can be
304   // controlled by the Arduino as master.
305   writeByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x22);
306   // Enable data ready (bit 0) interrupt
307   writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x01);
308   delay(100);
309 }
310
311
```

```
312 // Function which accumulates gyro and accelerometer data after device
313 // initialization. It calculates the average of the at-rest readings
      and then
314 // loads the resulting offsets into accelerometer and gyro bias
      registers.
315 void MPU9250::calibrateMPU9250(float * gyroBias, float * accelBias)
316 {
317   uint8_t data[12]; // data array to hold accelerometer and gyro x, y,
       z, data
318   uint16_t ii, packet_count, fifo_count;
319   int32_t gyro_bias[3]  = {0, 0, 0}, accel_bias[3] = {0, 0, 0};
320
321   // reset device
322   // Write a one to bit 7 reset bit; toggle reset device
323   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, READ_FLAG);
324   delay(100);
325
326   // get stable time source; Auto select clock source to be PLL
      gyroscope
327   // reference if ready else use the internal oscillator, bits 2:0 =
      001
328   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
329   writeByte(MPU9250_ADDRESS, PWR_MGMT_2, 0x00);
330   delay(200);
331
332   // Configure device for bias calculation
333   // Disable all interrupts
334   writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x00);
335   // Disable FIFO
336   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
337   // Turn on internal clock source
338   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
339   // Disable I2C master
340   writeByte(MPU9250_ADDRESS, I2C_MST_CTRL, 0x00);
341   // Disable FIFO and I2C master modes
342   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x00);
343   // Reset FIFO and DMP
344   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x0C);
345   delay(15);
346
347   // Configure MPU6050 gyro and accelerometer for bias calculation
348   // Set low-pass filter to 188 Hz
349   writeByte(MPU9250_ADDRESS, CONFIG, 0x01);
350   // Set sample rate to 1 kHz
351   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
352   // Set gyro full-scale to 250 degrees per second, maximum
      sensitivity
```

```
353   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00);
354   // Set accelerometer full-scale to 2 g, maximum sensitivity
355   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
356
357   uint16_t  gyrosensitivity  = 131;   // = 131 LSB/degrees/sec
358   uint16_t  accelsensitivity = 16384; // = 16384 LSB/g
359
360   // Configure FIFO to capture accelerometer and gyro data for bias
       calculation
361   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x40);  // Enable FIFO
362   // Enable gyro and accelerometer sensors for FIFO  (max size 512
       bytes in
363   // MPU-9150)
364   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x78);
365   delay(40);  // accumulate 40 samples in 40 milliseconds = 480 bytes
366
367   // At end of sample accumulation, turn off FIFO sensor read
368   // Disable gyro and accelerometer sensors for FIFO
369   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
370   // Read FIFO sample count
371   readBytes(MPU9250_ADDRESS, FIFO_COUNTH, 2, &data[0]);
372   fifo_count = ((uint16_t)data[0] << 8) | data[1];
373   // How many sets of full gyro and accelerometer data for averaging
374   packet_count = fifo_count / 12;
375
376   for (ii = 0; ii < packet_count; ii++)
377   {
378     int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
379     // Read data for averaging
380     readBytes(MPU9250_ADDRESS, FIFO_R_W, 12, &data[0]);
381     // Form signed 16-bit integer for each sample in FIFO
382     accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1]  );
383     accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3]  );
384     accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5]  );
385     gyro_temp[0]  = (int16_t) (((int16_t)data[6] << 8) | data[7]  );
386     gyro_temp[1]  = (int16_t) (((int16_t)data[8] << 8) | data[9]  );
387     gyro_temp[2]  = (int16_t) (((int16_t)data[10] << 8) | data[11]);
388
389     // Sum individual signed 16-bit biases to get accumulated signed
      32-bit
390     // biases.
391     accel_bias[0] += (int32_t) accel_temp[0];
392     accel_bias[1] += (int32_t) accel_temp[1];
393     accel_bias[2] += (int32_t) accel_temp[2];
394     gyro_bias[0]  += (int32_t) gyro_temp[0];
395     gyro_bias[1]  += (int32_t) gyro_temp[1];
396     gyro_bias[2]  += (int32_t) gyro_temp[2];
```

```
397   }
398   // Sum individual signed 16-bit biases to get accumulated signed 32-
        bit biases
399   accel_bias[0] /= (int32_t) packet_count;
400   accel_bias[1] /= (int32_t) packet_count;
401   accel_bias[2] /= (int32_t) packet_count;
402   gyro_bias[0]  /= (int32_t) packet_count;
403   gyro_bias[1]  /= (int32_t) packet_count;
404   gyro_bias[2]  /= (int32_t) packet_count;
405
406   // Sum individual signed 16-bit biases to get accumulated signed 32-
        bit biases
407   if (accel_bias[2] > 0L)
408   {
409     accel_bias[2] -= (int32_t) accelsensitivity;
410   }
411   else
412   {
413     accel_bias[2] += (int32_t) accelsensitivity;
414   }
415
416   // Construct the gyro biases for push to the hardware gyro bias
        registers,
417   // which are reset to zero upon device startup.
418   // Divide by 4 to get 32.9 LSB per deg/s to conform to expected bias
        input
419   // format.
420   data[0] = (-gyro_bias[0] / 4  >> 8) & 0xFF;
421   // Biases are additive, so change sign on calculated average gyro
        biases
422   data[1] = (-gyro_bias[0] / 4)       & 0xFF;
423   data[2] = (-gyro_bias[1] / 4  >> 8) & 0xFF;
424   data[3] = (-gyro_bias[1] / 4)       & 0xFF;
425   data[4] = (-gyro_bias[2] / 4  >> 8) & 0xFF;
426   data[5] = (-gyro_bias[2] / 4)       & 0xFF;
427
428   // Push gyro biases to hardware registers
429   writeByte(MPU9250_ADDRESS, XG_OFFSET_H, data[0]);
430   writeByte(MPU9250_ADDRESS, XG_OFFSET_L, data[1]);
431   writeByte(MPU9250_ADDRESS, YG_OFFSET_H, data[2]);
432   writeByte(MPU9250_ADDRESS, YG_OFFSET_L, data[3]);
433   writeByte(MPU9250_ADDRESS, ZG_OFFSET_H, data[4]);
434   writeByte(MPU9250_ADDRESS, ZG_OFFSET_L, data[5]);
435
436   // Output scaled gyro biases for display in the main program
437   gyroBias[0] = (float) gyro_bias[0] / (float) gyrosensitivity;
438   gyroBias[1] = (float) gyro_bias[1] / (float) gyrosensitivity;
```

```
439   gyroBias[2] = (float) gyro_bias[2] / (float) gyrosensitivity;
440
441   // Construct the accelerometer biases for push to the hardware
       accelerometer
442   // bias registers. These registers contain factory trim values which
       must be
443   // added to the calculated accelerometer biases; on boot up these
       registers
444   // will hold non-zero values. In addition, bit 0 of the lower byte
       must be
445   // preserved since it is used for temperature compensation
       calculations.
446   // Accelerometer bias registers expect bias input as 2048 LSB per g,
       so that
447   // the accelerometer biases calculated above must be divided by 8.
448
449   // A place to hold the factory accelerometer trim biases
450   int32_t accel_bias_reg[3] = {0, 0, 0};
451   // Read factory accelerometer trim values
452   readBytes(MPU9250_ADDRESS, XA_OFFSET_H, 2, &data[0]);
453   accel_bias_reg[0] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
454   readBytes(MPU9250_ADDRESS, YA_OFFSET_H, 2, &data[0]);
455   accel_bias_reg[1] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
456   readBytes(MPU9250_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
457   accel_bias_reg[2] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
458
459   // Define mask for temperature compensation bit 0 of lower byte of
460   // accelerometer bias registers
461   uint32_t mask = 1uL;
462   // Define array to hold mask bit for each accelerometer bias axis
463   uint8_t mask_bit[3] = {0, 0, 0};
464
465   for (ii = 0; ii < 3; ii++)
466   {
467     // If temperature compensation bit is set, record that fact in
       mask_bit
468     if ((accel_bias_reg[ii] & mask))
469     {
470       mask_bit[ii] = 0x01;
471     }
472   }
473
474   // Construct total accelerometer bias, including calculated average
475   // accelerometer bias from above
476   // Subtract calculated averaged accelerometer bias scaled to 2048
       LSB/g
477   // (16 g full scale)
```

```
478   accel_bias_reg[0] -= (accel_bias[0] / 8);
479   accel_bias_reg[1] -= (accel_bias[1] / 8);
480   accel_bias_reg[2] -= (accel_bias[2] / 8);
481
482   data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
483   data[1] = (accel_bias_reg[0])      & 0xFF;
484   // preserve temperature compensation bit when writing back to
       accelerometer
485   // bias registers
486   data[1] = data[1] | mask_bit[0];
487   data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
488   data[3] = (accel_bias_reg[1])      & 0xFF;
489   // Preserve temperature compensation bit when writing back to
       accelerometer
490   // bias registers
491   data[3] = data[3] | mask_bit[1];
492   data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
493   data[5] = (accel_bias_reg[2])      & 0xFF;
494   // Preserve temperature compensation bit when writing back to
       accelerometer
495   // bias registers
496   data[5] = data[5] | mask_bit[2];
497
498   // Apparently this is not working for the acceleration biases in the
        MPU-9250
499   // Are we handling the temperature correction bit properly?
500   // Push accelerometer biases to hardware registers
501   writeByte(MPU9250_ADDRESS, XA_OFFSET_H, data[0]);
502   writeByte(MPU9250_ADDRESS, XA_OFFSET_L, data[1]);
503   writeByte(MPU9250_ADDRESS, YA_OFFSET_H, data[2]);
504   writeByte(MPU9250_ADDRESS, YA_OFFSET_L, data[3]);
505   writeByte(MPU9250_ADDRESS, ZA_OFFSET_H, data[4]);
506   writeByte(MPU9250_ADDRESS, ZA_OFFSET_L, data[5]);
507
508   // Output scaled accelerometer biases for display in the main
       program
509   accelBias[0] = (float)accel_bias[0] / (float)accelsensitivity;
510   accelBias[1] = (float)accel_bias[1] / (float)accelsensitivity;
511   accelBias[2] = (float)accel_bias[2] / (float)accelsensitivity;
512 }
513
514
515 // Accelerometer and gyroscope self test; check calibration wrt
       factory settings
516 // Should return percent deviation from factory trim values, +/- 14 or
        less
517 // deviation is a pass.
```

```
518 void MPU9250::MPU9250SelfTest(float * destination)
519 {
520   uint8_t rawData[6] = {0, 0, 0, 0, 0, 0};
521   uint8_t selfTest[6];
522   int32_t gAvg[3] = {0}, aAvg[3] = {0}, aSTAvg[3] = {0}, gSTAvg[3] = {
       0};
523   float factoryTrim[6];
524   uint8_t FS = 0;
525
526   // Set gyro sample rate to 1 kHz
527   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
528   // Set gyro sample rate to 1 kHz and DLPF to 92 Hz
529   writeByte(MPU9250_ADDRESS, CONFIG, 0x02);
530   // Set full scale range for the gyro to 250 dps
531   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 1 << FS);
532   // Set accelerometer rate to 1 kHz and bandwidth to 92 Hz
533   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, 0x02);
534   // Set full scale range for the accelerometer to 2 g
535   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 1 << FS);
536
537   // Get average current values of gyro and acclerometer
538   for (int ii = 0; ii < 200; ii++)
539   {
540     //Serial.print("BHW::ii = ");
541     //Serial.println(ii);
542     // Read the six raw data registers into data array
543     readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
544     // Turn the MSB and LSB into a signed 16-bit value
545     aAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
546     aAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
547     aAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
548
549     // Read the six raw data registers sequentially into data array
550     readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
551     // Turn the MSB and LSB into a signed 16-bit value
552     gAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
553     gAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
554     gAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
555   }
556
557   // Get average of 200 values and store as average current readings
558   for (int ii = 0; ii < 3; ii++)
559   {
560     aAvg[ii] /= 200;
561     gAvg[ii] /= 200;
562   }
563
```

```
564   // Configure the accelerometer for self-test
565   // Enable self test on all three axes and set accelerometer range to
        +/- 2 g
566   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0xE0);
567   // Enable self test on all three axes and set gyro range to +/- 250
        degrees/s
568   writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0xE0);
569   delay(25);  // Delay a while to let the device stabilize
570
571   // Get average self-test values of gyro and acclerometer
572   for (int ii = 0; ii < 200; ii++)
573   {
574     // Read the six raw data registers into data array
575     readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
576     // Turn the MSB and LSB into a signed 16-bit value
577     aSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
578     aSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
579     aSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
580
581     // Read the six raw data registers sequentially into data array
582     readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
583     // Turn the MSB and LSB into a signed 16-bit value
584     gSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1]) ;
585     gSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3]) ;
586     gSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5]) ;
587   }
588
589   // Get average of 200 values and store as average self-test readings
590   for (int ii = 0; ii < 3; ii++)
591   {
592     aSTAvg[ii] /= 200;
593     gSTAvg[ii] /= 200;
594   }
595
596   // Configure the gyro and accelerometer for normal operation
597   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
598   writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0x00);
599   delay(25);  // Delay a while to let the device stabilize
600
601   // Retrieve accelerometer and gyro factory Self-Test Code from
        USR_Reg
602   // X-axis accel self-test results
603   selfTest[0] = readByte(MPU9250_ADDRESS, SELF_TEST_X_ACCEL);
604   // Y-axis accel self-test results
605   selfTest[1] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_ACCEL);
606   // Z-axis accel self-test results
607   selfTest[2] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_ACCEL);
```

```
608   // X-axis gyro self-test results
609   selfTest[3] = readByte(MPU9250_ADDRESS, SELF_TEST_X_GYRO);
610   // Y-axis gyro self-test results
611   selfTest[4] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_GYRO);
612   // Z-axis gyro self-test results
613   selfTest[5] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_GYRO);
614
615   // Retrieve factory self-test value from self-test code reads
616   // FT[Xa] factory trim calculation
617   factoryTrim[0] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[0] - 1.0) ));
618   // FT[Ya] factory trim calculation
619   factoryTrim[1] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[1] - 1.0) ));
620   // FT[Za] factory trim calculation
621   factoryTrim[2] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[2] - 1.0) ));
622   // FT[Xg] factory trim calculation
623   factoryTrim[3] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[3] - 1.0) ));
624   // FT[Yg] factory trim calculation
625   factoryTrim[4] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[4] - 1.0) ));
626   // FT[Zg] factory trim calculation
627   factoryTrim[5] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
       selfTest[5] - 1.0) ));
628
629   // Report results as a ratio of (STR - FT)/FT; the change from
       Factory Trim
630   // of the Self-Test Response
631   // To get percent, must multiply by 100
632   for (int i = 0; i < 3; i++)
633   {
634     // Report percent differences
635     destination[i] = 100.0 * ((float)(aSTAvg[i] - aAvg[i])) /
       factoryTrim[i]
636                         - 100.;
637     // Report percent differences
638     destination[i + 3] = 100.0 * ((float)(gSTAvg[i] - gAvg[i])) /
       factoryTrim[i + 3]
639                             - 100.;
640   }
641 }
642
643
644 // Wire.h read and write protocols
645 uint8_t MPU9250::writeByte(uint8_t deviceAddress, uint8_t
```

```
      registerAddress,
646                               uint8_t data)
647 {
648   if (_csPin != NOT_SPI)
649   {
650     return writeByteSPI(registerAddress, data);
651   }
652   else
653   {
654     return writeByteWire(deviceAddress, registerAddress, data);
655   }
656 }
657
658 uint8_t MPU9250::writeByteSPI(uint8_t registerAddress, uint8_t
      writeData)
659 {
660   uint8_t returnVal;
661
662   SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST, SPI_MODE))
      ;
663   select();
664
665   SPI.transfer(registerAddress);
666   returnVal = SPI.transfer(writeData);
667
668   deselect();
669   SPI.endTransaction();
670 #ifdef SERIAL_DEBUG
671   Serial.print("MPU9250::writeByteSPI slave returned: 0x");
672   Serial.println(returnVal, HEX);
673 #endif
674   return returnVal;
675 }
676
677 uint8_t MPU9250::writeByteWire(uint8_t deviceAddress, uint8_t
      registerAddress,
678                                uint8_t data)
679 {
680   Wire.beginTransmission(deviceAddress);  // Initialize the Tx buffer
681   Wire.write(registerAddress);        // Put slave register address in
       Tx buffer
682   Wire.write(data);                   // Put data in Tx buffer
683   Wire.endTransmission();             // Send the Tx buffer
684   // TODO: Fix this to return something meaningful
685   return NULL;
686 }
687
```

```
688 // Read a byte from given register on device. Calls necessary SPI or
      I2C
689 // implementation. This was configured in the constructor.
690 uint8_t MPU9250::readByte(uint8_t deviceAddress, uint8_t
      registerAddress)
691 {
692   if (_csPin != NOT_SPI)
693   {
694     return readByteSPI(registerAddress);
695   }
696   else
697   {
698     return readByteWire(deviceAddress, registerAddress);
699   }
700 }
701
702 // Read a byte from the given register address from device using I2C
703 uint8_t MPU9250::readByteWire(uint8_t deviceAddress, uint8_t
      registerAddress)
704 {
705   uint8_t data; // `data` will store the register data
706
707   // Initialize the Tx buffer
708   Wire.beginTransmission(deviceAddress);
709   // Put slave register address in Tx buffer
710   Wire.write(registerAddress);
711   // Send the Tx buffer, but send a restart to keep connection alive
712   Wire.endTransmission(false);
713   // Read one byte from slave register address
714   Wire.requestFrom(deviceAddress, (uint8_t) 1);
715   // Fill Rx buffer with result
716   data = Wire.read();
717   // Return data read from slave register
718   return data;
719 }
720
721 // Read 1 or more bytes from given register and device using I2C
722 uint8_t MPU9250::readBytesWire(uint8_t deviceAddress, uint8_t
      registerAddress,
723                                uint8_t count, uint8_t * dest)
724 {
725   // Initialize the Tx buffer
726   Wire.beginTransmission(deviceAddress);
727   // Put slave register address in Tx buffer
728   Wire.write(registerAddress);
729   // Send the Tx buffer, but send a restart to keep connection alive
730   Wire.endTransmission(false);
```

```
731
732   uint8_t i = 0;
733   // Read bytes from slave register address
734   Wire.requestFrom(deviceAddress, count);
735   while (Wire.available())
736   {
737     // Put read results in the Rx buffer
738     dest[i++] = Wire.read();
739   }
740
741   return i; // Return number of bytes written
742 }
743
744 uint8_t MPU9250::readBytes(uint8_t deviceAddress, uint8_t
      registerAddress,
745                            uint8_t count, uint8_t * dest)
746 {
747   if (_csPin == NOT_SPI)  // Read via I2C
748   {
749     return readBytesWire(deviceAddress, registerAddress, count, dest);
750   }
751   else  // Read using SPI
752   {
753     return readBytesSPI(registerAddress, count, dest);
754   }
755 }
756
757 // Read the WHOAMI (WIA) register of the AK8963
758 // TODO: This method has side effects
759 uint8_t MPU9250::ak8963WhoAmI_SPI()
760 {
761   uint8_t response, oldSlaveAddress, oldSlaveRegister, oldSlaveConfig;
762   // Save state
763   oldSlaveAddress  = readByteSPI(I2C_SLV0_ADDR);
764   oldSlaveRegister = readByteSPI(I2C_SLV0_REG);
765   oldSlaveConfig   = readByteSPI(I2C_SLV0_CTRL);
766 #ifdef SERIAL_DEBUG
767   Serial.print("Old slave address: 0x");
768   Serial.println(oldSlaveAddress, HEX);
769   Serial.print("Old slave register: 0x");
770   Serial.println(oldSlaveRegister, HEX);
771   Serial.print("Old slave config: 0x");
772   Serial.println(oldSlaveConfig, HEX);
773 #endif
774
775   // Set the I2C slave addres of AK8963 and set for read
776   response = writeByteSPI(I2C_SLV0_ADDR, AK8963_ADDRESS | READ_FLAG);
```

```
777   // I2C slave 0 register address from where to begin data transfer
778   response = writeByteSPI(I2C_SLV0_REG, 0x00);
779   // Enable 1-byte reads on slave 0
780   response = writeByteSPI(I2C_SLV0_CTRL, 0x81);
781   delayMicroseconds(1);
782   // Read WIA register
783   response = writeByteSPI(WHO_AM_I_AK8963 | READ_FLAG, 0x00);
784
785   // Restore state
786   writeByteSPI(I2C_SLV0_ADDR, oldSlaveAddress);
787   writeByteSPI(I2C_SLV0_REG, oldSlaveRegister);
788   writeByteSPI(I2C_SLV0_CTRL, oldSlaveConfig);
789
790   return response;
791 }
```

Listing A.2 – Arquivo de funções do *Driver*